SRI International

Final Technical Report • 22 May 2000
Scientific Technical Papers

A Formal Methods Workbench For Critical Systems

Contract Number: F49620-95-C-0044
SRI Project ECU-P07108

Prepared by:
John Rushby
Computer Science Laboratory

Prepared for:
Department of the Air Force
Air Force Office of Scientific Research
801 N. Randolph St.
Arlington, Virginia 22203-1977

Approved:
Patrick Lincoln, Director
Computer Science Laboratory

William S. Mark, Vice-President
Information and Computing Sciences

333 Ravenswood Avenue • Menlo Park, CA 94025-3493 • (650) 859-2000

REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-00-

0240

'88

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 28, 1998	Final Technical Report 15 Jun 95 to 14 Jun 99

4. TITLE AND SUBTITLE
A Formal Methods Workbench for Critical Systems

5. FUNDING NUMBERS
F49620-95-C-0044

6. AUTHOR(S)
John Rushby

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Science Laboratory
SRI International
Menlo Park, CA 91025

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
801 N. Randolph St, Rm 732
Arlington, VA 22203-1977

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
F49620-95-C-0044

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The research performed in this project enhanced the PVS formal verification system to provide improved support for the development and assurance of fault-tolerant and safety-critical systems. Thc capabilities developed are freely available in the publicly distributed version of PVS.

In addition, new and more systematic methods were developed for verifying certain kinds of critical systems (including self-stabilizing ones), and an exciting new application of formal methods to the problems of automation surprises and mode confusions in advanced cockpits was pioneered.

Finally, new fault-tolerant algorithms and architectures were developed, and some significant demonstration applications of formal methods were performed.

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Abstract

The research performed in this project enhanced the PVS formal verification system to provide improved support for the development and assurance of fault-tolerant and safety-critical systems. The capabilities developed are freely available in the publicly distributed version of PVS.

In addition, new and more systematic methods were developed for verifying certain kinds of critical systems (including self-stabilizing ones), and an exciting new application of formal methods to the problems of automation surprises and mode confusions in advanced cockpits was pioneered.

Finally, new fault-tolerant algorithms and architectures were developed, and some significant demonstration applications of formal methods were performed.

Summary of Results Achieved

Strong typechecking is an efficient way to identify many problems in formal specifications just as it is for programs. For programming languages, there are limits on the power of typechecking because it is expected that this should be performed by a fast deterministic algorithm. In a formal verification environment, however, where a powerful theorem prover is available, the power of typechecking can be considerably increased by extending the type system so that typechecking can make use of the theorem prover. PVS includes an extension of this kind called "predicate subtyping." A comprehensive exposition of the utility of this mechanism was developed under this project and published in the leading software engineering journal [19].

Tabular specifications of various kinds are widely advocated and used for critical systems. As part of this project the PVS type system was further enhanced to incorporate a `table` construct that can support many of the tabular notations in widespread use [5]. This capability of PVS has been used by engineers at Ontario Hydro in the certification of the shutdown system for the Darlington nuclear power station and by engineers at NASA in analysis of requirements for modifications to the flight software of the Space Shuttle. Because tables are fully integrated in PVS, the analysis supported goes beyond simple typechecking and extends to model checking and full deductive verification of systems specified using tables [7].

To make the benefits and advantages of formal methods in general, and enhancements such as these in particular, more widely known, the project undertook many presentations at major international conferences and developed several expository papers. These include [6, 10, 11, 12, 14, 22].

Enhanced typechecking is a potent way to detect errors, and it has the advantage that the required theorem proving can often be completely automated. For full verification, however, the difficulty of developing the required proofs (and strengthening the invariants used) has been a major obstacle to widespread adoption. An exciting new approach avoids some of these difficulties by using automated property-preserving abstraction to a reduced description that can be analyzed by algorithmic methods such as model checking. A very efficient and powerful method for constructing such abstractions automatically was developed as part of this project and is incorporated in PVS [20]. A protocol example whose verification previously required two months of interactive development is solved automatically using this new capability of PVS in a few seconds.

Other substantial methodological advances developed in the project include a "lazy" approach to compositional development and verification [21] and, in joint work with Professor Amir Pnueli of the Weizmann Institute, an approach to fair synchronous systems [8].

The project also investigated improved architectures and algorithms for fault-tolerant systems. In traditional fault-masking architectures based on state-machine replication, there is tension between the desire to exclude permanently faulty components as soon as they can be detected, and to avoid excluding components that are merely transiently faulty. An architecture that allows components to be placed "on probation" so that they can remain synchronized and current without being able to compromise the output of the system was developed as part of this project [13]. This architecture provides a way to exploit the improved diagnosis algorithms that were verified in related work [23].

Another tension exists in the design of consensus algorithms for state-machine replication, where one would like to exploit the improved error detection of cryptographically signed messages, without being totally dependent on the cryptographic assumption. The project developed novel algorithms that exploit the full power of signed messages, but that still work as well as unsigned message algorithms if the cryptographic assumptions are violated [1]. The analysis reported in this paper also made novel use of model checking to examine the behavior of algorithms beyond their worst-case bounds.

A particularly attractive fault-tolerant architecture for critical control applications is the Time Triggered Architecture (TTA) of Hermann Kopetz that is being adopted in the automotive industry and for some aircraft controls applications. The project developed and formally verified a mapping between synchronous algorithms and time-triggered implementations suitable for TTA [15]. The project also developed a very attractive group membership algorithm for TTA-like architectures [3]. This algorithm achieves full agreement and self-diagnosis with only one bit of overhead per message. Formal verification of this algorithm posed a formidable challenge that was overcome in subsequent work [17].

Self-stabilizing algorithms are attractive for many critical applications, particularly those that must tolerate transient faults. Their formal verification has posed significant challenges. The project developed a reasonably straightforward verification for one of the classic self-stabilizing algorithms (Dijkstra's algorithm for mutual exclusion in a ring) [9], and then devel-

Papers Published

These papers are all available from `http://www.csl.sri.com/fm-papers.html`.

[1] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In Ravishankar K. Iyer, Michele Morganti, W. Kent Fuchs, and Virgil Gligor, editors, *Dependable Computing for Critical Applications—5*, Volume 10 of IEEE Computer Society *Dependable Computing and Fault Tolerant Systems*, pages 139–157, Champaign, IL, September 1995.

[2] Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, Volume 1633 of Springer-Verlag *Lecture Notes in Computer Science*, Trento, Italy, July 1999.

[3] Shmuel Katz, Pat Lincoln, and John Rushby. Low-overhead time-triggered group membership. In Marios Mavronicolas and Philippas Tsigas, editors, *11th International Workshop on Distributed Algorithms (WDAG '97)*, Volume 1320 of Springer-Verlag *Lecture Notes in Computer Science*, pages 155–169, Saarbrücken Germany, September 1997.

[4] Sandeep Kulkarni, John Rushby, and N. Shankar. A case study in component-based mechanical verification of fault-tolerant programs. In *ICDCS Workshop on Self-Stabilizing Systems*, pages 33–40, IEEE Computer Society, Austin, TX, June 1999.

[5] Sam Owre, John Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, Volume 1217 of Springer-Verlag *Lecture Notes in Computer Science*, pages 366–383, Enschede, The Netherlands, April 1997.

[6] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, Volume 1641 of Springer-Verlag *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, October 1998.

[7] Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Available, with specification files, at `http://www.csl.sri.com/csl-95-12.html`. Also published as NASA Contractor Report 201729.

[8] Amir Pnueli, N. Shankar, and Eli Singerman. Fair synchronous transition systems and their liveness proofs. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 1486 of Springer-Verlag *Lecture Notes in Computer Science*, pages 198–209, Lyngby, Denmark, September 1998.

[9] Shaz Qadeer and Natarajan Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paul de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)*. pages 424–443, Shelter Island, NY, June 1998.

[10] John Rushby. Mechanizing formal methods: Opportunities and challenges. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, Volume 967 of Springer-Verlag *Lecture Notes in Computer Science*, pages 105–113, Limerick, Ireland, September 1995.

[11] John Rushby. Automated deduction and formal methods. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, Volume 1102 of Springer-Verlag *Lecture Notes in Computer Science*, pages 169–183, New Brunswick, NJ, July/August 1996.

[12] John Rushby. Mechanized formal methods: Progress and prospects. In V. Chandru and V. Vinay, editors, *16th Conference on the Foundations of Software Technology and Theoretical Computer Science*, Volume 1180 of Springer-Verlag *Lecture Notes in Computer Science*, pages 43–51, Hyderabad, India, December 1996.

[13] John Rushby. Reconfiguration and transient recovery in state-machine architectures. In *Fault Tolerant Computing Symposium 26*, pages 6–15, IEEE Computer Society, Sendai, Japan, June 1996.

[14] John Rushby. Calculating with requirements. In *3rd IEEE International Symposium on Requirements Engineering*, pages 144–146, IEEE Computer Society, Annapolis, MD, January 1997.

[15] John Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September/October 1999.

[16] John Rushby. Using model checking to help discover mode confusions and other automation surprises. In Denis Javaux and Véronique De Keyser, editors, *Proceedings of the 3rd Workshop on Human Error, Safety, and System Development (HESSD'99)*, University of Liege, Belgium, June 1999.

[17] John Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, Chicago, IL, July 2000. To appear.

[18] John Rushby, Judith Crow, and Everett Palmer. An automated method to detect potential mode confusions. In *18th AIAA/IEEE Digital Avionics Systems Conference*, St Louis, MO, October 1999.

[19] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[20] Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [2], pages 443–454.

[21] N. Shankar. Lazy compositional verification. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (Revised lectures from International Symposium COMPOS'97)*, Volume 1536 of Springer-Verlag *Lecture Notes in Computer Science*, pages 541-564, Bad Malente, Germany, September 1997.

[22] Natarajan Shankar. Unifying verification paradigms. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 1135 of Springer-Verlag *Lecture Notes in Computer Science*, pages 22–39, Uppsala, Sweden, September 1996.

[23] Chris J. Walter, Patrick Lincoln, and Neeraj Suri. Formally verified on-line diagnosis. *IEEE Transactions on Software Engineering*, 23(11):684–721, November 1997.

Subtypes for Specifications: Predicate Subtyping in PVS

John Rushby Sam Owre N. Shankar

Abstract—**A specification language used in the context of an effective theorem prover can provide novel features that enhance precision and expressiveness. In particular, typechecking for the language can exploit the services of the theorem prover. We describe a feature called "predicate subtyping" that uses this capability and illustrate its utility as mechanized in PVS.**

Keywords—**Formal methods, specification languages, type systems, subtypes, typechecking, consistency, PVS**

I. Introduction

FOR programming languages, type systems and their associated typecheckers are intended to ensure the absence of certain undesirable behaviors during program execution [1]. The undesired behaviors generally include untrapped errors such as adding a boolean to an integer, and may (e.g., in Java) encompass security violations. If the language is "type safe," then all programs that can exhibit these undesired behaviors will be rejected during typechecking.

Execution is not a primary concern for specification languages—indeed, they usually admit constructs, such as quantification over infinite domains or equality at higher types, that are not effectively computable—but typechecking can still serve to reject specifications that are erroneous or undesirable in some way. For example, a minimal expectation for specifications is that they should be consistent: an inconsistent specification is one from which some statement and its negation can both be derived; such a specification necessarily allows *any* property to be derived and thus fails to say anything useful at all. The first systematic type system (now known as the "Ramified Theory of Types") was developed by Russell [2] to avoid the inconsistencies in naïve set theory, and a simplified form of this system (the "Simple Theory of Types," due to Ramsey [3] and Church [4]) provides the foundation for most specification languages based on higher-order logic. If a specification uses no axioms (beyond those of the logic itself), then typechecking with respect to such a type system guarantees consistency. However, the consistency of specifications (such as 2 in Section III) that include extra-logical axioms cannot be checked algorithmically in general, so the best that a typechecker can do in the presence of axioms is to guarantee "conservative extension" of the other parts of the specification (i.e., roughly speaking, that it does not introduce any new inconsistencies).

Since their presence weakens the guarantees provided by typechecking, it is desirable to limit the use of axioms and to prefer those parts of the specification language for which typechecking ensures conservative extension. Unfortunately, those parts are usually severely limited in expressiveness and convenience, often being restricted to quantifier-free (though possibly recursive) definitions that have a strongly constructive flavor; such specifications may resemble implementations rather than statements of required properties, and proofs about them may require induction rather than ordinary quantifier reasoning. Thus, a worthwhile endeavor in the design of type systems for specification languages is to increase the expressiveness and convenience of those constructions for which typechecking can guarantee conservative extension, so that the drawbacks to a definitional style are reduced and resort to axioms is needed less often.

In developing type systems for specification languages, we can consider some design choices that are not available for programming languages. In particular, a specification language is meant to be part of an environment that includes an effective theorem prover, so it is feasible to contemplate that typechecking can rely on general theorem proving, and not be restricted to the trivially decidable properties that are appropriate for programming languages.

"Predicate subtypes" are one example of the opportunities that become available when typechecking can use theorem proving.[1] Predicate subtypes can be used to check statically for violations such as division by zero or out-of-bounds array references, and can also express more sophisticated consistency requirements. Typechecking with respect to predicate subtypes is done by proof obligation (verification condition) generation.

In the following sections we will use simple examples to explain what predicate subtypes are, and to demonstrate their utility in a variety of situations. The examples illustrating the use of predicate subtypes are all drawn from the PVS specification language [6].

II. Predicate Subtypes

Types in specification languages are often interpreted as sets of values, and this leads to a natural association of subtype with subset: one type is a subtype of another if

The authors are with the Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA

This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931. This is an expanded version of an invited paper presented at the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering and Sixth European Software Engineering Conference, Zurich, Switzerland, September 1997. Springer-Verlag Lecture Notes in Computer Science, volume 1301, pages 4–19.

[1] Another is consistency checking for tabular specifications [5].

the set interpreting the first is a subset of that interpreting the second. In this treatment (found, for example, in Mizar [7]) the natural numbers are a subtype of the integers, but the subtyping relation does not characterize those integers that are natural numbers. *Predicate* subtypes provide such a tightly bound characterization by associating a predicate or property with the subtype. For example, the natural numbers are the subtype of the integers characterized by the predicate "greater than or equal to zero." Predicate subtypes can help make specifications more succinct by allowing information to be moved into the types, rather than stated repeatedly in conditional formulas. For example, instead of

```
∀(i, j:int):i ≥ 0 ∧ j ≥ 0 ⊃ i+j ≥ i
```

(where we use ⊃ for logical implication) we can say

```
∀(i, j:nat):i+j ≥ i
```

because `i ≥ 0` and `j ≥ 0` are recorded in the type `nat` for `i` and `j`.

Theorem proving can be required in typechecking some constructions involving predicate subtypes. For example, if `half` is a function that requires an `even` number (defined as one equal to twice some integer) as its argument, then the formula

```
∀(i:int):half(i+i+2) = i+1
```

is well-typed only if we can prove that the integer expression `i+i+2` satisfies the predicate for the subtype `even`: that is, if we can discharge the following proof obligation.

```
∀(i:int):∃(j:int):i+i+2 = 2×j                                    1
```

Predicate subtypes seem a natural idea and often appear, in inchoate form, in informal mathematics. Similar ideas are also seen in formalized specification notations where, for example, the datatype invariants of VDM [8, Chapter 5] have much in common with predicate subtypes. However, datatype invariants are part of VDM's mechanisms for specifying operations in terms of pre- and post-conditions on a state, rather than part of the type system for its logic. Predicate subtypes are fully supported as part of a specification logic by the Nuprl [9], ABEL [10], Raise [11], Veritas [12], and PVS [6] verification systems. Predicate subtypes arose independently in these systems (in PVS, they came from its predecessor, EHDM, whence they were introduced from the ANNA notation [13] by Friedrich von Henke, who was involved in the design of both), and there are differences in their uses and mechanization. In Nuprl, for example, all typechecking relies on theorem proving, whereas in PVS there is a firm distinction between conventional typechecking (which is performed algorithmically) and the proof obligations (they are called Type Correctness Conditions, or TCCs in PVS) engendered by certain uses of predicate subtyping.

The circumstances in which proof obligations are generated, and other properties of predicate subtypes are described in the remainder of this paper. The examples use PVS notation, which is briefly introduced in the following section.

PVS and its Notation for Predicate Subtypes

PVS is a higher-order logic in which the simple theory of types is augmented by dependent types and predicate subtypes. Type constructors include functions, tuples, records, and abstract data types (freely generated recursive types) such as trees and lists. A large collection of standard theories is provided in libraries and in the PVS "prelude" (which is a built-in library). The PVS system includes an interactive theorem prover that can be customized with user-written "strategies" (similar to tactics and tacticals in LCF-style provers), and that provides rather powerful automation in the form of decision procedures (e.g., for ground equality and linear arithmetic over both integers and reals) integrated with a rewriter [14, 15].

As noted, some constructions involving predicate subtypes generate TCCs (proof obligations); these are not decidable in general as there are no constraints on the predicates used to induce subtypes. However, many of the TCCs encountered in practice do fall within a class that is decided by the automated procedures of PVS. In other cases, the user must develop suitable proofs interactively; this arrangement provides the flexibility of arbitrary type constraints without loss of automation on the decidable ones. Proof of TCCs can be postponed, but the system keeps track of all undischarged proof obligations and the affected theories and theorems are marked as incomplete.

A PVS specification is a collection of theories. Each theory takes a list of theory parameters and provides a list of declarations or definitions for variables, individual constants, types, and formulas. Types in PVS are built starting with uninterpreted types and primitive interpreted ones, such as `bool`, `int` (integer), `nat` (natural number), and various other numeric types. Record types are given as a list of label/type pairs: for example, `[# age:nat, years_employed:nat #]`. Tuples, such as `[nat, bool]`, are similar to records except that fields are accessed by the order of their appearance rather than by labels. Function types are introduced by specifying their domain and range types: for example, binary arithmetic operations such as addition and multiplication have the type `[[real, real]→real]`, which can also be written as `[real, real→real]`.

Functions (and predicates, which are simply functions with range type `bool`) can be defined using λ-notation, so that the predicate that recognizes even integers can be written as follows (it is a PVS convention that predicates have names ending in "?").[2]

```
even?:[int→bool] = λ(i:int):∃(j:int):i = 2×j
```

However, the following "applicative" form is exactly equivalent and is generally preferred.

```
even?(i:int):bool = ∃(j:int):i = 2×j
```

The strictness of the type hierarchy ensures that the principle of comprehension is sound in higher-order logic: that is,

[2]For ease of reading, the typeset rendition of PVS is used here; PVS can generate this automatically using its LaTeX-printer. PVS uses the Gnu Emacs editor as a front end and its actual input is presented in ASCII.

predicates and sets can be regarded as essentially equivalent.[3] PVS therefore also allows set notation for predicates, so that the following definition is equivalent to the previous two.

```
even?: [int→bool] = {i:int | ∃(j:int): i = 2×j}
```

Viewed as a predicate, the test that an integer **x** is even is written **even?(x)**; viewed as a set it is written **x ∈ even?**. These are notational conveniences; semantically, the two forms are equivalent.

Predicates induce a subtype over their domain type; this subtype can be specified using set notation (overloading the previously introduced use of set notation to specify predicates), or by enclosing a predicate in parentheses. Thus, the following are all semantically equivalent, and denote the type of even integers.

```
even: TYPE = {i:int | ∃(j:int): i=2×j}
even: TYPE = (even?)
even: TYPE = (λ(i:int):∃(j:int): i=2×j)
even: TYPE = ({i:int | ∃(j:int): i=2×j})
```

III. Discovering Errors with Predicate Subtypes

PVS makes no *a priori* assumptions about the cardinality of the sets that interpret its types: they may be empty, finite, or countably or uncountably infinite. When an uninterpreted constant is declared, however, we need to be sure that its type is not empty (otherwise we have a contradiction). This cannot be checked algorithmically when the type is a predicate subtype, so an "existence TCC" is generated that obliges the user to prove the fact.[4] Thus the constant declaration

```
c: even
```

generates the following proof obligation, which requires nonemptiness of the **even** type to be demonstrated.

```
c_TCC1: OBLIGATION ∃(x: even): TRUE
```

The existence TCC is a potent detector of erroneous specifications when higher (i.e., function and predicate) types are involved, as the following example illustrates.

Suppose we wish to specify a function that returns the minimum of a set of natural numbers presented as its argument. Definitional specifications for this function are likely to be rather unattractive—certainly involving a recursive definition and possibly some concrete choice about how sets are to be represented. An axiomatic specification, on the other hand, seems very straightforward: we simply state that the minimum is a member of the given set, and no larger than any other member of the set. In PVS this could be written as follows.

[3] All members of a set are of the same type in higher-order logic; this notion of "set" differs from that used in set theory where $\{a, \{a\}\}$, for example, is a valid set.

[4] If the constant is interpreted (e.g., **c: even = 2**), then the proof obligation is to show that its value satisfies the corresponding predicate (e.g., **∃ (j: int): 2=2×j**).

```
min(s:setof[nat]):nat                                          2

simple_ax:AXIOM
  ∀(s:setof[nat]):min(s)∈s ∧ ∀(n:nat):n∈s ⊃ min(s)≤n
```

Here, the first declaration gives the "signature" of the function, stating that it takes a set of natural numbers as its argument and returns a natural number as its value. The axiom **simple_ax** then formalizes the informal specification in the obvious way, and seems innocuous enough. However, as many readers will have noticed, this axiom harbors an inconsistency: it states that the function returns a member of its argument **s**—but what if **s** is empty?

How could predicate subtypes alert us to this inconsistency? Well, as noted earlier, sets and predicates are equivalent in higher-order logic, so that a set **s** of natural numbers is also a predicate on the natural numbers, and thereby induces the predicate subtype **(s)** comprising those natural numbers that satisfy (or, viewed as a set, are members of) **s**. Thus we can modify the signature of our **min** function to specify that it returns, not just a natural number, but one that is a member of the set supplied as its argument.[5]

```
min(s:setof[nat]):(s)
```

Now this declaration is asserting the existence of a function having the given signature and, in higher-order logic, functions are just constants of "higher" type. Because we have asserted the existence of a constant, we need to ensure that its type is nonempty, so PVS generates the following TCC.

```
min_TCC1: OBLIGATION ∃(x:[s:setof[nat]→(s)]):TRUE
```

Inspection, or fruitless experimentation with the theorem prover, should convince us that this TCC is unprovable and, in fact, false.[6] We are thereby led to the realization that our original specification is unsound, and the **min** function must not be required to return a member of the set supplied as its argument when that set is empty.

We have a choice at this point: we could either return to the original signature for the **min** function in 2 and weaken its axiom appropriately, or we could strengthen the signature still further so that the function simply cannot be applied to empty sets. The latter choice best exploits the capabilities of predicate subtyping, so that is the one used here. The predicate that tests a set of natural numbers for nonemptiness is written **nonempty?[nat]** in PVS, so the type of nonempty sets of natural numbers is written **(nonempty?[nat])**, and the strict signature for a **min** function can be specified as follows.

```
min(s:(nonempty?[nat])):(s)
```

[5] This is an example of a "dependent" type: it is dependent because the *type* of one element (here, the range of the function) depends on the *value* of another (here, the argument supplied to the function). Dependent typing is essential to derive the full utility of predicate subtyping. It is discussed in more detail in Section VII.

[6] A function type is nonempty if its range type is nonempty or its domain type is empty. Here the domain type is nonempty (be careful not to confuse emptiness of the domain *type*, **setof[nat]**, with emptiness of the *argument* **s**), so we need to be sure that the range type, **(s)**, is also nonempty—which it is not, when **s** is empty.

This declaration generates the following TCC

```
min_TCC1: OBLIGATION ∃(x: [s: (nonempty?[nat]) → (s)]): TRUE
```

which can be discharged by instantiating x with the choice function for nonempty types that is built-in to PVS.[7]

With its signature taken care of, we can now return to the axiom that specifies the essential property of the min function. First, notice that the first conjunct in the axiom simple_ax shown in [2] is unnecessary now that this constraint is enforced in the range type of the function. Next, notice that the implication in the second conjunct can be eliminated by changing the quantification so that n ranges over only members of s, rather than over all natural numbers. This leads to the following more compact axiom.

```
min_ax: AXIOM ∀(s:(nonempty?[nat])),(n:(s)): min(s) ≤ n
```

Satisfied that this specification is correct (as indeed it is), we might be tempted to make the "obvious" next step and define a max function dually.

```
max(s:(nonempty?[nat])):(s)
max_ax: AXIOM ∀(s:(nonempty?[nat])),(n:(s)): max(s) ≥ n
```

This apparently small extension introduces another inconsistency: for what if the set s is infinite? Infinite sets of natural numbers have a minimum element, but not a maximum. Let us see how predicate subtypes could help us avoid this pitfall.

Using predicate subtyping, we can eliminate the axiom max_ax and add the property that it expresses to the range type of the max function as follows.

```
max(s:(nonempty?[nat])):{x:(s) | ∀(n:(s)):x ≥ n}        3
```

This causes PVS to generate the following TCC to ensure nonemptiness of the function type specified for max.

```
max_TCC1: OBLIGATION
  ∃(x1: [s: (nonempty?[nat]) → {x: (s) | ∀(n: (s)): x ≥ n}]): TRUE
```

Observe that by moving what was formerly specified by an axiom into the specification of the range type, we are using PVS's predicate subtyping to mechanize generation of proof-obligations for the axiom satisfaction problem.

We begin the proof of this TCC by instantiating x1 with the (built-in) choice function choose, applied to the predicate {x: (s) | ∀(n: (s)): x ≥ n} that appears as the range type.

```
(INST + "λ(s:(nonempty?[nat])):
           choose({x:(s) | ∀(n:(s)):x ≥ n})")
```

PVS proof commands are given in Lisp syntax; the first term identifies the command (here "INST" for instantiate), the second generally indicates those formulas in the sequent (see below) to which the command should be applied (+ means "any formula in the consequent part of the sequent"), and any required PVS text is enclosed in quotes. The next two proof commands

[7] We need to demonstrate the existence of a function that takes a nonempty set of natural numbers as its argument and returns a member of that set as its value. Choice functions, which are discussed in Section IV, have exactly this property.

```
(GRIND :IF-MATCH NIL)
(REWRITE "forall_not")
```

then reduce the TCC to the following proof goal. (s!1 and x!1 are the Skolem constants corresponding to the quantified variables s and x in the original formula).

```
[-1]    x!1 ≥ 0                                          4
[-2]    s!1(x!1)
  |-------
{1}     ∃(x:(s!1)):∀(n:(s!1)):x ≥ n
```

This is a "sequent," which is the manner in which PVS presents the intermediate stages in a proof. In general, there is a collection of "antecedent" formulas (here two) above the sequent line (|-------), and a collection (here, only one) of "consequents" below; the sequent is true if the conjunction of formulas above the line implies the disjunction of formulas below (if there are no formulas below the line then we need a contradiction among those above). PVS proof commands transform the current sequent to one or more simpler (we hope) sequents whose truth implies the original one. The three proof commands shown earlier respectively instantiate an existentially quantified variable (INST), perform quantifier elimination, definition expansion, and invoke decision procedures (GRIND; the annotation :IF-MATCH NIL instructs the prover not to attempt instantiation of variables), and apply a rewrite rule (REWRITE; the rule concerned comes from the PVS prelude and changes a $\forall \ldots \neg \ldots$ above the line into an $\exists \ldots$ below the line, which makes it easier to read). Once again, inspection, or fruitless experimentation with the theorem prover, should persuade us that the goal [4] is unprovable (we are asked to prove that any nonempty set of natural numbers has a largest element) and thereby reveals the flaw in our specification.

The flaw revealed in max might cause us to examine a specification for min given in the same form as [3] to check that it does not have the same problem. This min specification generates a TCC that reduces to a goal similar to [4] (with $\leq$ substituted for $\geq$ in the consequent) but, unlike the max case, this goal is true, and can be proved by appealing to the well-foundedness (i.e., absence of infinite descending chains) of the less-than ordering on natural numbers.

With the significance of well-foundedness now revealed to us, we might attempt to specify a generic min function: one that is defined over any type, with respect to a well-founded ordering on that type.

```
minspec[T:TYPE, <:(well_founded?[T])]:THEORY                5
BEGIN
  IMPORTING equalities[T]

  min((s:(nonempty?[T]))):{x:(s) | ∀(i:(s)):x<i ∨ x=i}
END minspec
```

This specification introduces a general min function in the context of a theory parameterized by an arbitrary (and possibly empty) type T, and a well-founded ordering < over that type. Notice how predicate subtyping is used in the formal parameter list of this theory to specify that < must be well-founded (the predicate well_founded? is defined in

to programming languages seems a worthwhile research endeavor that might generalize the benefits of extended static checking, while also providing information that could be useful to an optimizing compiler.

Subtypes of a different, "structural," kind are sometimes used in type systems for programming languages to account for issues arising in object-oriented programs [1]. In particular, a record type `A` that contains fields in addition to those of a record type `B` is regarded as a subtype of `B`. The intuition behind this kind of subtyping is rather different than the "subtypes as subsets" intuition. Here, the idea is that a subtype is an elaboration of a type, so that anywhere a value of a certain type is required, it should be acceptable to supply a value of a subtype of that type (e.g., a function that requires "points" should find a "colored point" acceptable). Structural subtypes are characterized by having a canonical coercion from the subtype to the supertype (e.g., by dropping the extra fields from a record) so that a supertype operation can be applied by means of this coercion. Using this approach, some operations can be structural subtype polymorphic—meaning that they apply uniformly to all structural subtypes of a given type. When these ideas are applied to functions, they lead to the "normal" or *covariant* subtyping on range types, but *contravariant* subtyping on domain types.

We know of no specification language that provides structural subtyping, still less combines it with predicate subtyping. The difficulty is that reasoning about equality is problematical in the presence of structural record subtypes and contravariant or covariant function subtyping.

In PVS the type of the equality relation in an equation `x = y` can be determined simply by considering the types of `x` and `y` (it is equality on their least common supertype). For example, in the expression `x/y` $\times$ `y = x`, where `x` and `y` are natural numbers, it is equality on rationals (because the division operator coerces the left hand expression to be rational). This means that if `x = y` and `y = z` are type-correct, then so is `x = z`, and it is true if the other two are. Now consider contravariant subtyping on function domains. This allows `abs = id[nat]`, where `abs: [int→nat]` is the absolute value function on the integers and `id[T]` is the identity function on type `T` (here, the naturals). This follows by promoting `abs` to its contravariant supertype `[nat→nat]` and taking equality on that type. It also allows `id[nat] = id[int]` by the same reasoning, and transitivity of equality might lead us to conclude `abs = id[int]`. But there is no reason to invoke contravariant subtyping in this final equation, since both functions have the same domain (we do need to use covariant subtyping on the range), so the equality is on the type `[int→int]`, and the equation is false. (The equation is true if equality is interpreted on `[nat→nat]` but, as noted, there is no reason to assign this type on the basis of the arguments appearing in the expression.)

Because of this difficulty in contravariant subtyping on function domains,[15] and in order to allow equals to be freely substituted, we have chosen to allow function subtyping in PVS only when the domains are equal. PVS does extend subtyping covariantly over the range types of functions (e.g., `[nat → nat]` is a subtype of `[nat → int]`) and over the positive parameters to abstract data types (e.g., list of `nat` is a subtype of list of `int`), since these cases do not present problems. We consider predicate subtyping to be a basic element of a specification language—whereas structural subtyping and subtyping on function domains are largely syntactic conveniences that we prefer to handle by mechanisms such as conversions (see also [29]), rather than incorporate them into the type system. Nonetheless, combining some structural subtyping (e.g., for records) with predicate subtyping is an interesting topic for research.

[15] Covariant subtyping on domains presents difficulties, too: it seems to require partial functions.

XI. Conclusion

We have illustrated a few circumstances where predicate subtypes contribute to the clarity and precision of a specification, to the identification of errors, and to the automation provided in analysis of specifications and in theorem proving. There are many other circumstances where predicate subtypes provide benefit, and they have been used to excellent effect by several users of PVS (see, for example, the PVS bibliography [30] and the links from its Web page). We hope the examples we have presented do convey the value of predicate subtyping in specification languages, and suggest their possible utility in programming languages.

Acknowledgments

Paul Jackson provided many suggestions that have improved the presentation.

References

Papers by SRI authors are generally available from `http://www.csl.sri.com/fm.html`.

[1] Luca Cardelli, "Type systems," in *Handbook of Computer Science and Engineering*, chapter 103, pp. 2208–2236. CRC Press, 1997. Available at `http://www.research.digital.com/SRC`.

[2] Bertrand Russell, "Mathematical logic as based on the theory of types," in *From Frege to Gödel*, Jean van Heijenoort, Ed., pp. 150–182. Harvard University Press, Cambridge, MA, 1967, First published 1908.

[3] F. P. Ramsey, "The foundations of mathematics," in *Philosophical Papers of F. P. Ramsey*, D. H. Mellor, Ed., chapter 8, pp. 164–224. Cambridge University Press, Cambridge, UK, 1990, Originally published in *Proceedings of the London Mathematical Society*, 25, pp. 338–384, 1925.

[4] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940.

[5] Sam Owre, John Rushby, and N. Shankar, "Integration in PVS: Tables, types, and model checking," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, Ed Brinksma, Ed., Enschede, The Netherlands, Apr. 1997, vol. 1217 of *Lecture Notes in Computer Science*, pp. 366–383, Springer-Verlag.

[6] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 107–125, Feb. 1995.

[7] Piotr Rudnicki, "An overview of the MIZAR project," in *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, Båstad, Sweden, June 1992, pp. 311–330, Available at `http://web.cs.ualberta.ca/~piotr/Mizar/MizarOverview.ps`.

[8] Cliff B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, second edition, 1990.

[9] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith, *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[10] Ole-Johan Dahl and Olaf Owe, "On the use of subtypes in ABEL (revised version)," Tech. Rep. 206, Department of Informatics, University of Oslo, Oct. 1995.

[11] The RAISE Language Group, *The RAISE Specification Language*, BCS Practioner Series. Prentice-Hall International, Hemel Hempstead, UK, 1992.

[12] F. Keith Hanna, Neil Daeche, and Mark Longley, "Specification and verification using dependent types," *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 949–964, Sept. 1989.

[13] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe, *ANNA: A Language for Annotating Ada Programs*, vol. 260 of *Lecture Notes in Computer Science*, Springer-Verlag, 1987.

[14] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas, "PVS: Combining specification, proof checking, and model checking," In Alur and Henzinger [31], pp. 411–414.

[15] John Rushby, "Automated deduction and formal methods," In Alur and Henzinger [31], pp. 169–183.

[16] David Cyrluk, Patrick Lincoln, and N. Shankar, "On Shostak's decision procedure for combinations of theories," in *Automated Deduction—CADE-13*, M. A. McRobbie and J. K. Slaney, Eds., New Brunswick, NJ, July/Aug. 1996, vol. 1104 of *Lecture Notes in Artificial Intelligence*, pp. 463–477, Springer-Verlag.

[17] H. Barringer, J. H. Cheng, and C. B. Jones, "A logic covering undefinedness in program proofs," *Acta Informatica*, vol. 21, pp. 251–269, 1984.

[18] Michael J. Beeson, *Foundations of Constructive Mathematics*, Ergebnisse der Mathematik und ihrer Grenzgebiete; 3. Folge · Band 6. Springer-Verlag, 1985.

[19] David Lorge Parnas, "Predicate logic for software engineering," *IEEE Transactions on Software Engineering*, vol. 19, no. 9, pp. 856–862, Sept. 1993.

[20] William M. Farmer, "A partial functions version of Church's simple theory of types," *Journal of Symbolic Logic*, vol. 55, no. 3, pp. 1269–1291, Sept. 1990.

[21] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer, "IMPS: An interactive mathematical proof system," *Journal of Automated Reasoning*, vol. 11, no. 2, pp. 213–248, Oct. 1993.

[22] Leslie Lamport and Lawrence C. Paulson, "Should your specification language be typed?," SRC Research Report 147, Digital Systems Research Center, Palo Alto, CA, May 1997. Available at `http://www.research.digital.com/SRC`.

[23] J. H. Cheng and C. B. Jones, "On the usability of logics which handle partial functions," in *Proceedings of the Third Refinement Workshop*, Carroll Morgan and J. C. P. Woodcock, Eds. 1990, pp. 51–69, Springer-Verlag Workshops in Computing.

[24] Savi Maharaj and Juan Bicarregui, "On the verification of VDM specification and refinement with PVS," in *12th IEEE International Conference on Automated Software Engineering: ASE '97*, Incline Village, NV, Nov. 1997, IEEE Computer Society, pp. 280–289.

[25] Mark Saaltink, "The Z/EVES system," in *ZUM '97: The Z Formal Specification Notation; 10th International Conference of Z Users*, Reading, UK, Apr. 1997, vol. 1212 of *Lecture Notes in Computer Science*, pp. 72–85, Springer-Verlag.

[26] Mark Saaltink, "Domain checking Z specifications," in *LFM' 97: Fourth NASA Langley Formal Methods Workshop*, C. Michael Holloway and Kelly J. Hayhurst, Eds., Hampton, VA, Sept. 1997, NASA Langley Research Center, NASA Conference Publication 3356, pp. 185–192. Available at `http://atb-www.larc.nasa.gov/Lfm97/proceedings/`.

[27] Steven M. German, "Automating proofs of the absence of common runtime errors," in *Proceedings, 5th ACM Symposium on the Principles of Programming Languages*, Tucson, AZ, Jan. 1978, pp. 105–118.

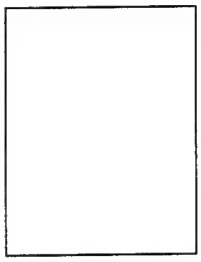
[28] David L. Detlefs, "An overview of the Extended Static Checking system," in *First Workshop on Formal Methods in Software Practice (FMSP '96)*, San Diego, CA, Jan. 1996, Association for Computing Machinery, pp. 1–9.

[29] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov, "Inheritance as implicit coercion," *Information and Computation*, vol. 93, no. 1, pp. 172–221, July 1991.

[30] John Rushby, *PVS Bibliography*, Menlo Park, CA, Constantly updated; available at `http://www.csl.sri.com/pvs-bib.html`.

[31] Rajeev Alur and Thomas A. Henzinger, Eds., *Computer-Aided Verification, CAV '96*, vol. 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.

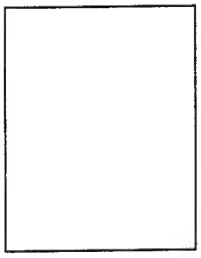
The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.



John Rushby received B.Sc. and Ph.D. degrees in computing science from the University of Newcastle upon Tyne in 1971 and 1977, respectively. He joined the Computer Science Laboratory of SRI International in 1983, and served as its director from 1986 to 1990; he currently manages its research program in formal methods and dependable systems. Prior to joining SRI, he held academic positions at the Universities of Manchester and Newcastle upon Tyne in England.

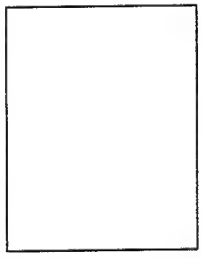
Dr. Rushby leads the projects developing and applying PVS. His research interests center on the use of formal methods for problems in design and assurance for dependable systems. He is the author of the section on formal methods for the FAA Digital Systems Validation Handbook.

He is a member of the IEEE, the Association for Computing Machinery, the American Institute of Aeronautics and Astronautics, and the American Mathematical Society. He is an associate editor for these Transactions, and a member of the editorial board for the journal "Formal Aspects of Computing."



Sam Owre received his B.S. in mathematics from Stevens Institute of Technology in 1975, and an M.A. in mathematics from the University of California, Los Angeles in 1978. He is a Senior Software Engineer in the Computer Science Laboratory at SRI International, where for the past eight years he has devoted most of his waking hours to the development of the PVS and EHDM verification systems. Prior to that he worked in a number of AI-related research projects at Advanced Decision Systems, and before that he built yet another verification system (described in the February 1987 issue of this journal) while working at Sytek Inc. He has coauthored a number of papers on formal methods.

He is a member of the IEEE, the Association for Computing Machinery, the Association for Symbolic Logic, and the American Mathematical Society.



Natarajan Shankar received a B.Tech. degree in electrical engineering from the Indian Institute of Technology, Madras in 1980, and a Ph.D. in computer science from the University of Texas at Austin in 1986. He has been a computer scientist with the Computer Science Laboratory at SRI International since 1989. Prior to joining SRI, he was a research associate with the Stanford University Computer Science Department. His interests include formal methods, automated reasoning, metamathematics, and linear logic. He co-developed SRI's PVS specification language and verification system. His book *Metamathematics, Machines, and Gödel's Proof* was published by Cambridge University Press and is now in its second printing.

Dr. Shankar is a member of the Association for Computing Machinery, the Association for Symbolic Logic, and the IFIP Working Group 2.3 on programming methodology. He is a member of the editorial board for the journal "Formal Methods in Systems Design."

the PVS prelude). A proof obligation to check satisfaction of this requirement will be generated whenever the theory is instantiated. Observe that the specification has been adjusted a little to separate the $<$ and $=$ cases that were combined into $\leq$ for the special case of natural numbers.

Typechecking this specification results in the following TCC, requiring us to demonstrate that the function type asserted for `min` is nonempty.

```
min_TCC1: OBLIGATION                                                    6
  ∃(x1: [s: (nonempty?[T])→{x: (s) | ∀(i: (s)): x < i ∨ x = i}]):
    TRUE
```

As before, we begin the proof of this TCC by instantiating it with the choice function `choose`, applied to the predicate `{x: (s) | ∀(i: (s)): x<i ∨ x=i}` that appears as the range type.

```
(INST + "λ(s:(nonempty?[T])):
     choose({x:(s) | ∀(i:(s)):x<i ∨ x=i})")
```

This discharges the original proof obligation, but `choose` requires its argument to be nonempty, so the prover generates a new TCC subgoal to establish this fact.

```
min_TCC1 (TCC):                                                         7

  |-------
{1}    ∀(s:(nonempty?[T])):
           nonempty?[(s)]({x:(s) | ∀(i:(s)):x<i ∨ x=i})
```

This is asking us to demonstrate the existence of a minimal element for any nonempty set `s` (more precisely, it is asking us to demonstrate the nonemptiness of the set of all such minimal elements). Now the type specified for $<$ requires it to be a well-founded ordering, and we can introduce this knowledge into the proof by the command `(TYPEPRED "<")`. The command `(GRIND :IF-MATCH NIL)` then produces the following simplified sequent.

```
{-1}    s!1(x!1)
{-2}    ∀(p:pred[T]):
          (∃(y:T):p(y))
             ⊃ (∃(y:(p)):(∀(x:(p)):(¬ x < y)))
{-3}    ∀(x:(s!1)):¬ ∀(i:(s!1)):x < i ∨ x = i
  |-------
```

Here, the formula `{-2}` is expressing the well-foundedness of $<$; instantiating the variable `p` with `s!1` and giving a few more interactive commands, we arrive at the following sequent (this is one of two subgoals generated; the other is trivial).

```
[-1]    s!1(x!1)
  |-------
{1}    i!1 < y!1
{2}    y!1 < i!1
{3}    y!1 = i!1
```

For the specialized `min` function on natural numbers, the decision procedures completed the proof at this point, but here we recognize that this goal is not true in general, and we need the additional assumption that the relation $<$ be trichotomous (i.e., one of the three consequents must hold, as they do on the natural numbers). Once again, predicate subtypes have led us to discover an error in our specification. We can exit the prover, modify the specification [5] to stipulate that the theory parameter $<$ must be of type `well_ordered?[T]` (a well-ordering is one that is both well-founded and trichotomous) and rerun the proof of the TCC. This time we are successful.

Given the generic theory, we can recover `min` on the natural numbers by the instantiation `min[nat,<]`. Because of the subtype constraint specified for the second formal parameter to the theory, PVS generates a TCC requiring us to establish that $<$ on the natural numbers is a well-ordering. This is easily done, but `min[nat,>]` correctly generates a false TCC (this theory instantiation is equivalent to our previous attempt to specify a `max` function on the naturals). However, the TCC for `min[{i: int | i<0},>]` (i.e., the max function on the negative integers) is true and provable.

The examples in this section illustrate how a uniform check for nonemptiness of the type declared for a constant leads to the discovery of several quite subtle errors in the formulation of an apparently simple specification. In our experience, the same benefit accrues in larger specifications.

IV. Automating Proofs with Predicate Subtypes

A theorem prover is needed to discharge the proof obligations engendered by predicate subtyping. Conversely, however, predicate subtypes provide information that can actively assist a theorem prover. In this section we illustrate two ways in which predicate subtypes can help automate proofs, beginning with the use of very precise range types for functions.

A couple of the proofs in the previous section used the "choice function" `choose`. PVS actually has two choice functions defined in its prelude. The first, `epsilon`, is simply Hilbert's ε operator.

```
epsilons [T: NONEMPTY_TYPE]: THEORY
BEGIN
  p: VAR setof[T]

  epsilon(p): T

  epsilon_ax: AXIOM (∃x: x ∈ p) ⊃ epsilon(p) ∈ p
END epsilons
```

Given a set `p` over a nonempty type `T`, `epsilon(p)` is some member of `p`, if any such exist, otherwise it is just some value of type `T`. (The `VAR` declaration for `p` simply allows us to omit its type from the declarations where it is used; PVS formulas are implicitly universally quantified over their free variables.)

If `p` is constrained to be nonempty, then we can give the following specification for an `epsilon_alt` function, which is simply `epsilon` specialized to this situation (note that `T` need not be specified as `NONEMPTY_TYPE` in this case).

```
choice [T: TYPE]: THEORY
BEGIN
  p: VAR (nonempty?[T])

  epsilon_alt(p): T

  epsilon_alt_ax: AXIOM epsilon_alt(p) ∈ p
END choice
```

The new choice function **epsilon_alt** is similar to the built-in function **choose**, but if we return to the proof of **min_TCC1** (recall [6]) but use **epsilon_alt** in place of **choose**, we find that in addition to the subgoal [7], we are presented with the following.

```
  |-------                                                          [8]
{1} ∀(s: (nonempty?[T])):∀(i: (s)):
      epsilon_alt[(s)]({x: (s)|∀(i: (s)): x<i ∨ x=i}) < i
       ∨epsilon_alt[(s)]({x: (s)|∀(i: (s)): x<i ∨ x=i}) = i
```

This subgoal requires us to prove that the value of **epsilon_alt** satisfies the predicate supplied as its argument; it can be discharged by appealing to **epsilon_alt_ax**, but the proof takes several steps and generates a further subgoal that is similar to [7] (and proved in the same way). How is it that the choice function **choose** avoids all this work that **epsilon_alt** seems to require?

The explanation is found in the definition of **choose**.

```
p: VAR (nonempty?[T])

choose(p): (p)
```

This very economical definition uses a predicate subtype to specify the property previously stated in **epsilon_alt_ax**: namely, that the value of **choose(p)** is a member of **p**.[8] But because the fact is stated in a subtype and is directly bound to the range type of **choose**, it is immediately available to the theorem prover—which is therefore able to discharge the equivalent to [8] internally.

A further use of subtypes to assist automation of proofs involves *encapsulation*. Below is the specification for the cardinality of a finite set.[9]

```
S: VAR finite_set
n: VAR nat

inj_set(S): (nonempty?[nat]) =
   {n | ∃(f: [(S)→below(n)]): injective?(f)}

Card(S): nat = min(inj_set(S))
```

Here, **inj_set(S)** is the set of natural numbers **n** for which there is an injection from **S** to the initial segment of natural numbers smaller than **n** (the predicate **injective?** is defined in [16] in Section VII). **Card(s)** is then defined as the least such **n**.

This construction has the advantage of being definitional, and therefore demonstrably sound, but it is inconvenient to work with. Consequently, the PVS library provides numerous lemmas that are derived from the definition but are more suitable for automated reasoning. Unfortunately, however, the automated prover strategies may sometimes choose to open up the definition of **Card** when it would be better to rewrite with the lemmas.

[8]The full definition is actually **choose(p): (p) = epsilon(p)**; this additionally specifies that **choose(p)** returns the same value as **epsilon(p)**, which is useful in specifications that use both **epsilon** and **choose**.

[9]This technique was developed by Ricky Butler and Paul Miner of NASA Langley, and the specification is from the **new_finite_sets** library that was largely developed by them and is distributed with the current version of PVS.

One way to abstract away the definition of cardinality is to use predicate subtypes to encapsulate it in the type of a new cardinality function.

```
card(S): {n: nat | n = Card(S)}
```

This implicitly defines **card** because the range type is a singleton. The lemmas can then be stated in terms of **card**, which is used as the main cardinality function, and automated prover strategies can be used safely because there is no definition of **card** for them to open up inappropriately.

Whereas the previous section demonstrated the utility of predicate subtypes in detecting errors in specifications, the examples in this section demonstrate their utility in improving the automation of proofs. When properties are specified axiomatically, it can be quite difficult to automate selection and instantiation of the appropriate axioms during a proof (unless they have special forms, such as rewrite rules). Properties expressed as predicate subtypes on the type of a term are, however, intimately bound to that term, and it is therefore relatively easy for a theorem prover to locate and instantiate the property automatically. Predicate subtypes also provide a form of encapsulation, so that specifications can be written in a style that prevents the theorem prover from opening up certain definitions.

V. Enforcing Invariants with Predicate Subtypes

Consider a specification for a city phone book. Given a name, the phone book should return the set of phone numbers associated with that name; there should also be functions for adding, changing, and deleting phone numbers. Here is the beginning of a suitable specification in PVS, giving only the basic types, and a function for adding a phone number **p** to those recorded for name **n** in phone book **B**.

```
names, phone_numbers: TYPE
phone_book: TYPE = [names→setof[phone_numbers]]
B: VAR phone_book
n: VAR names
p: VAR phone_numbers

add_number(B, n, p): phone_book = B WITH [(n) := B(n)∪{p}]
...
```

Here, the **WITH** construction is PVS notation for function overriding: **B WITH [(n) := B(n)∪{p}]** is a function that has the same values as **B**, except that at **n** it has the value **B(n)∪{p}**.

Now suppose we wish to enforce a constraint that the sets of phone numbers associated with different names should be disjoint. We can easily do this by introducing the **unused_number** predicate and modifying the **add_number** function as follows.

```
unused_number(B, p): bool = ∀(n: names): ¬ p ∈ B(n)                  [9]

add_number(B, n, p): phone_book =
  IF unused_number(B, p) THEN B WITH [(n) := B(n)∪{p}]
                         ELSE B ENDIF
```

If we had specified other functions for updating the phone book, they would need to be modified similarly.

But where in this modified specification does it say explicitly that different names must have disjoint sets of phone numbers? And how can we check that our specifications of updating functions such as **add_number** preserve his property? Both deficiencies are easily overcome with a predicate subtype: we simply change the type **phone_book** to the following.

```
phone_book: TYPE =                                                    10
      {B: [ names → setof[phone_numbers]] |
           ∀(n, m: names): n ≠ m ⊃ disjoint?(B(n), B(m))}
```

This states exactly the property we require. Furthermore, typechecking the specification [9] now causes the following proof obligation to be generated. Similar proof obligations would be generated for any other functions that update the phone book.

```
add_number_TCC1: OBLIGATION                                           11
    ∀(B, n, p): unused_number(B, p)
        ⊃ ∀(r, m: names): r ≠ m
            ⊃ disjoint?(B WITH [(n) := B(n)∪{p}](r),
                        B WITH [(n) := B(n)∪{p}](m))
```

This proof obligation, which is discharged by three commands to the PVS theorem prover, requires us to prove that a **phone_book B** (having the disjointness property), will satisfy the disjointness property after it has been updated by the **add_number** function.

The proof obligation in [11] arises for the same reason as the one in [1]: a value of the parent type has been supplied where one of a subtype is required, so a proof obligation is generated to establish that the value satisfies the predicate of the subtype concerned. Here, the body of the definition given for **add_number** in [9] has type **[names → setof[phone_numbers]]**, which is the parent type given for **phone_book** in [10], and so the proof obligation [11] is generated to check that it satisfies the appropriate predicate.

Observe how this uniform check on the satisfaction of predicate subtypes automatically generates the proof obligations necessary to ensure that the functions on a data type (here, **phone_book**) preserve an invariant. In the absence of such automation, we would have to formulate the appropriate proof obligations manually (a tedious and error-prone process), or construct a proof obligation generator for this one special purpose. The following section describes how the same mechanism can alleviate difficulties caused by partial functions.

VI. Avoiding Partial Functions with Predicate Subtypes

Functions are primitive and total in higher-order logic, whereas in set theory they are constructed as sets of pairs and are generally partial. There are strong advantages in theorem proving from adopting the first approach: it allows use of congruence closure as a decision procedure for equality over uninterpreted function symbols, which is essential for effective automation [16]. On the other hand, there are functions, such as division, that seem inherently partial and cause difficulty to this approach. One way out of the difficulty is introduce some artificial value for undefined terms such as $x/0$, but this is clumsy and has to be done carefully to avoid inconsistencies. Another approach introduces "undefined" as a truth value [17]; more sophisticated approaches use "free logics" in which quantifiers range only over defined terms (e.g., Beeson's logic of partial terms [18]; Parnas [19] and Farmer [20] have introduced logics similar to Beeson's[10]). Both approaches have the disadvantage of using nonstandard logics, with some attendant difficulties. These problems have led some to argue that the discipline of types can be too onerous in a specification language, and that untyped set theory is a better choice [22].

Predicate subtypes offer another approach. Many partial functions become total if their domains are specified with sufficient precision; applying a function outside its domain then becomes a type error, rather than something that has to be dealt with in the logic. Predicate subtypes provide the tool necessary to specify domains with suitable precision.

For example, division is a total function if it is typed so that its second argument must be nonzero. In PVS this can be specified as follows.

```
nonzero_real: TYPE = {x: real | x ≠ 0}                                12
/: [real, nonzero_real→real]
```

Now consider the well-formedness of following formula.

```
test: THEOREM ∀(x, y: real): x ≠ y ⊃ (x-y)/(y-x) = -1                 13
```

Subtraction is closed on the reals, so **x-y** and **y-x** are both reals. The second argument to the division function is required to have type **nonzero_real**; **real** is its parent type, so we have the proof obligation **(y-x) ≠ 0**, which is not true in general. However, the antecedent to the implication in [13] will be false when **x = y**, rendering the theorem true independently of the value of the improperly typed application of division. This leads to the idea that the proof obligation should take account of the context in which the application occurs, and should require only that the application is well-typed in circumstances where its value matters. In this case, a suitable, and easily proved, proof obligation is the following.

```
test_TCC1: OBLIGATION ∀(x, y: real): x ≠ y ⊃ (y-x) ≠ 0
```

This is, in fact, the TCC generated by PVS from the formula [13]. PVS imposes a left-to-right interpretation on formulas, and generates TCCs that ensure well-formedness under the logical context accumulated in that order. For example, the requirements for well-formedness of an implication $P \supset Q$ are that P be well-formed, and that Q be well-formed under the assumption that P is true; **if-then-else** is treated as two implications, and the rules for disjunctions $P \vee Q$ and conjunctions $P \wedge Q$ are similar to that for implication, except that for disjunctions Q must be shown well-formed under the assumption that P is

[10] Farmer's logic is used in the IMPS system [21]. IMPS generates proof obligations to ensure definedness during proofs that are similar to PVS's TCCs. However, because the properties required to discharge these are not bound to the types, many similar proof obligations can arise repeatedly during a single proof; IMPS mitigates this problem using caching.

false. Thus, PVS generates the same TCC as above when the formula in [13] is reformulated as follows.

```
test: THEOREM ∀(x, y:real): x = y ∨ (x-y)/(y-x) = -1
```

However, the accumulation of context in left-to-right order (which is sound, but conservative) causes PVS to generate the unprovable TCC (y-x) $\neq$ 0 for the following, logically equivalent, reformulation.

```
test: THEOREM ∀(x, y:real): (x-y)/(y-x) = -1 ∨ x = y    [14]
```

Another example of a partial function is the *subp* "challenge" from Cheng and Jones [23]. This function on integers is given by

$$subp(i,j) = \textbf{if } i = j \textbf{ then } 0 \textbf{ else } subp(i,j+1)+1 \textbf{ endif}$$

and is undefined if $i < j$ (when $i \geq j, subp(i,j) = i - j$).

As described in an earlier paper [6, Section III], this challenge is easily handled in PVS using dependent predicate subtyping to require that the second argument is no greater than the first. The function is then specified as follows.

```
subp((i:int),(j:int | j ≤ i)):RECURSIVE nat =
  IF i = j THEN 0 ELSE subp(i, j+1) + 1 ENDIF
MEASURE i-j
```

This generates three TCCs: one to ensure that the recursive call satisfies the type specified for the arguments, one to ensure that i-j in the MEASURE satisfies the predicate for nat, and another to establish termination using this measure. All three are discharged automatically by the PVS decision procedures.

The PVS formulation of subp is adequate for most purposes, but the following formula (from Maharaj and Bicarregui [24]) reveals a limitation.

```
subp_lemma: FORMULA                                    [15]
   ∀ i,j:nat: subp(i,j)=i-j ∨ subp(j,i)=j-i
```

This formula is true in some treatments of partial functions, but generates false TCCs and is unacceptable to PVS.

We have considered using symmetric rules for TCC generation so that examples such as [14] and [15] can be accepted: the left side of an expression would be typechecked in the context of the right side, as well as vice-versa, and the expression would be considered type-correct if either proof obligation can be discharged. However, we decided not to adopt this treatment for reasons of simplicity and efficiency. Since most specifications are written to be read from left to right (for the convenience of human readers), the conservatism of the left-to-right interpretation is seldom a problem in practice. (An exception is when PVS specifications are mechanically generated from some other representation.)

Predicate subtypes also yield an elegant treatment of recursive datatypes. The stack datatype, for example, can be specified as consisting of a constructor empty and another constructor push with accessors top and pop. In PVS, this is specified concisely as follows.

[10]The traditional notation for the second bound variable is (j: {j:int | j≤i}); PVS also allows the less redundant form used here.

```
stack [T: TYPE]: DATATYPE
BEGIN
  empty: empty?
  push (top: T, pop:stack): nonempty?
END stack
```

Each constructor defines a disjoint subtype of the datatype so that the top and pop operations are total on nonempty? stacks and it is type-incorrect to apply them to possibly empty? stacks. Thus, the following definition

```
doublepop(s: (nonempty?)): stack = pop(pop(s))
```

correctly generates the following unprovable and untrue TCC.

```
doublepop_TCC1: OBLIGATION
      ∀ (s: (nonempty?[T])): nonempty?[T](pop[T](s))
```

In our experience, use of predicate subtypes to render functions total is not onerous, and contributes clarity and precision to a specification; it also provides potent error detection. As another illustration of the latter, the "domain checking" for Z specifications provided by the Z/EVES system [25] has reportedly found errors in every Z specification examined in this way [26]. (Domain checking is similar to the use of predicate subtypes described in this section, but lacks the more general benefits of predicate subtyping.)

VII. Dependent Types and Higher-Order Subtypes

Predicate subtypes are useful for defining very refined type dependencies through dependent typing. We have already seen a few examples of dependent typing, such as some of the treatments of min in Section III and subp in the previous section. Here, we illustrate its use to constrain the fields of records to "reasonable" values, and to ensure a natural treatment of equality.

Dependent typing can be used to constrain the type of one field in a record according to the value of another field, as in the following example.

```
employee_record:TYPE =
                  [# age:nat, years_employed:upto(age) #]
```

This record declaration constrains the years_employed field to take values that are bounded above by the value of the age field (upto(n) is the type {i:nat | i≤n}). This type would thus rule out a record such as the following.

```
Jones: employee_record = (# age:=30, years_employed:=40 #)
```

The utility of the combination of dependent typing and predicate subtyping can be further illustrated by an example due to Carl Witty: the "implementation" of stacks as a record consisting of a size field and an array of elements of some type T. A simple formalization of this is the following record type.

```
stack_imp:TYPE = [# size:nat, elements:[nat→T] #]
```

The problem with this implementation is that two stacks that have the same size, say n, and agree on the first n values in the elements array, can still be unequal by disagreeing on irrelevant array values (i.e., those beyond size).[11]

[11]Note that equality on functions is extensional: two functions f and g of type [D→R] are equal iff for all x in D, $f(x) = g(x)$.

This makes it awkward to formulate correctly even such simple theorems as `pop(push(x, stack)) = stack`. The problem can be solved using dependent typing to reformulate the specification as follows.

```
stack_imp:TYPE=[# size:nat, elements:[below(size)→T] #]
```

With this refined typing, two stacks are equal when they have the same size and agree on the stack elements. Finite sequences are defined similarly in the PVS prelude.

PVS is a higher-order logic, meaning that functions can be applied to functions, and quantification can range over function types. Consequently, predicates can be defined over functions, and induce corresponding subtypes. For example, the following theory from the PVS prelude defines the predicates `injective?`, `surjective?` and `bijective?` over functions from `D` to `R`.[12]

```
functions [D, R: TYPE]: THEORY                                    16
 BEGIN
  f: VAR [D→R]
  x,x1,x2: VAR D
  y: VAR R
  injective?(f): bool = ∀ x1, x2: f(x1)=(x2) ⊃ x1=x2
  surjective?(f): bool = ∀ y: ∃ x: f(x) = y
  bijective?(f): bool = injective?(f) ∧ surjective?(f)
```

These induce corresponding subtypes, allowing declarations such as the following.

```
int2nat:(bijective?[int,nat]) =
      λ(i:int): IF i>0 THEN 2*i-1 ELSE -2*i ENDIF
```

By the standard mechanisms, this generates a TCC requiring a demonstration that the function `int2nat` is indeed a bijection between the integers and the naturals.

The combination of predicate subtyping, dependent typing, and higher-order types and subtypes is a powerful one. Higher-order subtypes can be used to introduce types such as order-preserving or order-inverting maps, and monotone predicate transformers. A drawback to some uses of predicate subtypes, however, is that large numbers of TCCs may be generated. In the next section we describe how this drawback can be overcome.

VIII. Judgements

The examples given in the preceding three sections illustrate the proof obligations that are generated when a term of a given type is provided where a subtype is expected. This can lead to a proliferation of many similar proof obligations. One way the PVS typechecker controls this proliferation is to check whether a new TCC is subsumed by earlier ones in the same theory: the TCC is suppressed if it is so subsumed. Although this does remove some duplicates, it is still possible to generate TCCs that differ only in some irrelevant contextual formulas.

An effective way to minimize—and often eliminate—the generation of trivially different TCCs for a given expression is for the user to state the needed proof obligation once and for all, and in its strongest form. In PVS, this is accomplished using *judgements*. The simplest form of judgement states that a given constant has a specific type. For example, we could give the judgement declaration

```
JUDGEMENT 2 HAS_TYPE even                                         17
```

This generates an immediate TCC to show that `2` is indeed even, but the typechecker can then make use of this fact to avoid generating similar TCCs in any context where the judgement is visible.

Judgements can also assert subtype constraints on the value returned by a function in terms of those on its arguments. Suppose we have the following formula declaration.

```
h: FORMULA ∀ (e:even): half(e+2) = e/2 + 1                        18
```

Recall that `half` requires an even argument. We would like the typechecker to recognize that the result of adding two even numbers is again even; this can be accomplished with the following judgement declaration.[13]

```
JUDGEMENT + HAS_TYPE [even, even → even]                          19
```

Such a judgement over a function type is interpreted as a *closure condition* equivalent to the following formula.

```
∀(e1, e2: even): even?(e1 + e2)
```

This is, in fact, the TCC generated by this judgement declaration.[14] The combination of the judgements 17 and 19 allows the PVS typechecker to determine immediately that the application of `half` in 18 is well-typed.

The final kind of judgement informs the typechecker of subtype relations that are not explicit in their constructions. For example, the types `nonzero_real` and `nonzero_rational` are defined as follows in the PVS prelude.

```
nonzero_real: NONEMPTY_TYPE = {r: real | r ≠ 0}
nonzero_rational: NONEMPTY_TYPE = {r: rational | r ≠ 0}
```

From this the typechecker can deduce the subtype relationship between `nonzero_rational` and `rational`, and hence also the type `real` (since `rational` is defined as a subtype of `real`), but it cannot deduce a subtyping relation between `nonzero_rational` and `nonzero_real`. With division typed as in 12 in Section VI, the following formula generates a TCC to show that `q` is nonzero.

```
div_lt_1: FORMULA ∃ (q:nonzero_rational):1/q = 2
```

Such proof obligations can be avoided if the desired subtype relation is stated explicitly using following judgement.

```
JUDGEMENT nonzero_rational SUBTYPE_OF nonzero_real
```

As with other judgement declarations, this immediately generates the necessary TCC and enlarges the collection of facts known to the typechecker, thereby reducing the number of TCCs generated subsequently.

[12] Since a tuple type may be supplied for `D`, this theory is fully general and can be instantiated for functions of any arity.

[13] This, along with a large number of similar judgements, is in the PVS prelude.

[14] A future version of PVS will allow such closure constraints to be stated more directly as: `JUDGEMENT +(e1,e2:even) HAS_TYPE even`

IX. Conversions

Conversions are functions that the typechecker can insert automatically whenever there is a type mismatch. Their purpose is to provide increased syntactic convenience in situations involving subtyping of higher types, but we introduce them by means of a simpler example.

```
c: [int→bool]
CONVERSION c
two: FORMULA 2
```

Here, since formulas must be of type boolean, the typechecker automatically invokes the conversion and changes the formula to `c(2)`. This is done internally, and is only visible to the user on explicit command and in the proof checker. (To avoid confusion, the typechecker warns the user if there is ever more than one applicable conversion.)

This simple kind of conversion has nothing to do with subtypes, but standard conversions **restrict** and **extend** do play an important role in handling subtyping on function types in PVS. The rule for subtyping of function types is straightforward for the range type, so that $[D \to R_1]$ is a subtype of $[D \to R_2]$ iff R_1 is a subtype of R_2. However, the treatment of domains is less obvious. The *covariant* approach regards $[D_1 \to R]$ as a subtype of $[D_2 \to R]$ iff D_1 is a subtype of D_2; conversely, the *contravariant* approach regards $[D_1 \to R]$ as a subtype of $[D_2 \to R]$ iff D_2 is a subtype of D_1. PVS makes a more restricted choice: the domains of two functions must be *equal* for them to have a subtyping relation (which is then determined by their range types). This choice keeps the semantics simple (see Section X), but prohibits some natural constructions. Consider the following, for example.

```
g: [int→int]
F: [[nat→int]→bool]
F_app: FORMULA F(g)
```

As this stands, `F_app` is not type-correct, because a function of type `[int→int]` is supplied where one of type `[nat→int]` is required, and PVS requires equality on domain types before a subtyping relation can be considered.

However, it is clear that `g` naturally induces a function from **nat** to **int** by simply restricting its domain. Such a domain restriction is achieved by the **restrict** conversion that is defined in the PVS prelude as follows.

```
restrict [T: TYPE, S: TYPE FROM T, R: TYPE]: THEORY
BEGIN
  f: VAR [T→R]
  s: VAR S
  restrict(f)(s): R = f(s)
  CONVERSION restrict
END restrict
```

The construction `S: TYPE FROM T` specifies that the actual parameter supplied for `S` must be a subtype of the one supplied for `T`. The specification states that **restrict(f)** is a function from `S` to `R` whose values agree with `f` (which is defined on the larger domain `T`). Using this approach, a type correct version of `F_app` can be written as `F(restrict[int,nat,int](g))`. This is, of course, inconvenient to read and write, so **restrict** is specified as a conversion, which allows the PVS typechecker to insert it automatically when needed, thereby providing much of the convenience of contravariant function subtyping in this circumstance.

It is not so obvious how to expand the domain of a function in the general case, so this approach does not work so automatically in the other direction. It does, however, work well for the important special case of sets (or, equivalently, predicates): a set on some type `S` can be extended naturally to one on a supertype `T` by assuming that the members of the type-extended set are just those of the original set. Thus, if **extend(s)** is the type-extended version of the original set `s`, we have **extend(s)(x) = s(x)** if `x` is in the subtype `S`, and **extend(s)(x) = false** otherwise. We can say that **false** is the "default" value for the type-extended function. Building on this idea, we arrive at the following specification for a general type-extension function.

```
extend [T: TYPE, S: TYPE FROM T, R: TYPE, d: R]: THEORY
BEGIN
  f: VAR [S→R]
  t: VAR T
  extend(f)(t): R = IF S_pred(t) THEN f(t) ELSE d ENDIF
END extend
```

The function **extend(f)** has type `[T→R]` and is constructed from the function `f` of type `[S→R]` (where `S` is a subtype of `T`) by supplying the default value `d` whenever its argument is not in `S` (`S_pred` is the *recognizer* predicate for `S`). Because of the need to supply the default `d`, this construction cannot be applied automatically as a conversion. However, as noted above, **false** is a natural default for functions with range type **bool** (i.e., sets and predicates), and the following theory establishes the corresponding conversion.

```
extend_bool [T: TYPE, S: TYPE FROM T]: THEORY
BEGIN
  CONVERSION extend[T, S, bool, false]
END extend_bool
```

In the presence of this conversion, the type-incorrect formula `B_app` in the following specification

```
b: [nat→bool]
B: [[int→bool]→bool]
B_app: FORMULA B(b)
```

is automatically modified by the typechecker to become `B(extend[int,nat,bool,false](b))`.

These examples illustrate the utility of conversions in bringing some of the convenience of contravariant function subtyping to the more restricted type system of PVS. Conversions are also useful (for example, in semantic encodings of temporal logics) in "lifting" operations to apply pointwise to sequences over their argument types.

X. Comparison with Subtypes in Programming Languages

We know of no programming language that provides predicate subtypes, although the annotations provided for "extended static checking" (proving the absence of runtime errors such as array bound violations) [27, 28] have some similarities. Bringing the benefits of predicate subtyping

Slightly expanded version of a paper presented at the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97), Enschede, The Netherlands, April 1997. Springer-Verlag Lecture Notes in Computer Science Vol. 1217, pp. 366–383.

Integration in PVS: Tables, Types, and Model Checking*

Sam Owre, John Rushby, Natarajan Shankar

Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA

Abstract. We have argued previously that the effectiveness of a verification system derives not only from the power of its individual features for expression and deduction, but from the extent to which these capabilities are integrated: the whole is more than the sum of its parts [20, 21]. Here, we illustrate this thesis by describing a simple construct for tabular specifications that was recently added to PVS. Because this construct integrates with other capabilities of PVS, such as typechecker-generated proof obligations, dependent typing, higher-order functions, model checking, and general theorem proving, it can be used for a surprising variety of purposes. We demonstrate this with examples drawn from hardware division algorithms and requirements specifications.

1 Introduction

Persuaded by the advocacy of David Parnas and others [15], we recently added a construct for tabular specification to PVS [12]. The construct generates proof obligations to ensure that the conditions labeling the rows and columns are disjoint and exclusive. This simple capability has been found useful by colleagues at NASA and Lockheed-Martin, who applied it in requirements analysis for Space Shuttle flight software [2, 18]. The capability becomes rather richer in the presence of dependent typing, and in this form it has been used to verify the accessible region in a quotient lookup table for SRT division [19]. When combined with other features of the PVS specification language, the table construct provides some of the attractive attributes of the TableWise [8] and SCR [6] specification methods. Because these constructions are performed in the context of a full verification system, we are able to use its theorem prover and model checker to establish invariant and reachability properties of the specifications concerned, and are able also to compose specifications described by separate tables and to establish refinement and equivalence relations between state machines specified in this manner.

* This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931.

2 Basic Tables

Tables can be a convenient way to specify certain kinds of functions. An example is the function $sign(x)$, which returns $-1, 0$, or 1 according to whether its integer argument is negative, zero, or positive. As a table, this can be specified as follows.

$$sign(x) = \begin{array}{|c|c|c|} \hline x < 0 & x = 0 & x > 0 \\ \hline -1 & 0 & +1 \\ \hline \end{array}$$

This is an example of a piecewise continuous function that requires definition by cases, and the tabular presentation provides two benefits.

- It makes the cases explicit, thereby allowing checks that none of them overlap and that all possibilities are considered.
- It provides a visually attractive presentation of the definition that eases comprehension.

The first of these benefits is a semantic issue that is handled in PVS by the `COND` construct; the second is a syntactic issue that is handled in PVS by the `TABLE` construct, which builds on `COND`.

Before we introduce these constructs, we should mention that the PVS specification language is a higher-order logic that supports both predicate subtypes and dependent types, and that the system provides strong assurances that definitional constructs (such as recursive function definitions) are conservative [13,14]. Some of the checks necessary to ensure type-correctness and conservative extension are not algorithmically decidable; in these cases, PVS generates Type Correctness Conditions (TCCs), which are obligations that must be discharged by theorem proving. PVS provides a powerful interactive theorem prover that includes decision procedures for linear arithmetic and other theories, and its default strategies are often able to discharge TCCs automatically; in more difficult cases, the user must guide the theorem prover interactively. Specifications with false TCCs are considered malformed and no meaning is ascribed to them. PVS allows proof obligations to be postponed, but keeps track of all unsatisfied obligations; a specification is not considered fully typechecked, and its theorems are considered provisional, until all TCCs have been proved.

2.1 The PVS `COND` Construct

Standard PVS language constructions for specification by cases are the traditional `IF-THEN-ELSE`, and a pattern matching `CASES` expression for enumerating over the constructors of an abstract data type. A `COND` construct has recently been added to these. Its general form is shown in 1, where the c_i are Boolean expressions and the e_i are values of some type t. (PVS has subtypes and overloading, so the types of the individual e_i must be "unified" to yield the common supertype t.) The keyword `ELSE` can be used in place of the final condition c_n. The construct can appear anywhere that a value of the type of t is allowed.

COND $c_1 \rightarrow e_1$, $c_2 \rightarrow e_2$, ... $c_n \rightarrow e_n$ ENDCOND	1
IF c_1 THEN e_1 ELSIF c_2 THEN e_2 ... ELSE e_n ENDIF	2

Exactly one of the c_i is required to be true; because PVS already supports proof obligations in the form of TCCs, it is easy to enforce this requirement by causing each COND to generate two TCCs as follows.

- *Disjointness* requires that each distinct c_i, c_j pair is disjoint.
- *Coverage* requires that the disjunction of all the c_i is true.

The coverage TCC is suppressed if the ELSE keyword is used; also the c_i, c_j component of the disjointness TCC is suppressed when e_i and e_j are syntactically identical.

A COND has meaning only if its TCCs are true, in which case the general COND expression of 1 is assigned the same meaning as (and is treated internally as) the IF-THEN-ELSE construction shown in 2. Notice that the condition c_n does not appear in the IF-THEN-ELSE translation: if this condition was given as an explicit ELSE in the COND, then the "fall through" default is exactly what is required; otherwise, the coverage TCC ensures that c_n is the negation of the disjunction of the other c_i, and the "fall through" is again correct. Because COND is treated internally as an IF-THEN-ELSE, reasoning involving COND requires no extensions to the PVS theorem prover.

Using COND, we can specify the *sign* function as follows.

```
signs: TYPE = { x: int | x >= -1 & x <= 1}
x: VAR int

sign_cond(x): signs =  COND
    x < 0  -> -1,
    x = 0  ->  0,
    x > 0  ->  1
ENDCOND
```

This generates the following TCCs, both of which are discharged by PVS's default strategy for TCCs in fractions of a second.

```
% Disjointness TCC generated (line 10) for
% COND x < 0 -> -1, x = 0 -> 0, x > 0 -> 1 ENDCOND
  sign_cond_TCC2: OBLIGATION  (FORALL (x: int):
    NOT (x < 0 AND x = 0)
     AND NOT (x < 0 AND x > 0)
      AND NOT (x = 0 AND x > 0));

% Coverage TCC generated (line 10) for
% COND x < 0 -> -1, x = 0 -> 0, x > 0 -> 1 ENDCOND
  sign_cond_TCC3: OBLIGATION (FORALL (x: int): x < 0 OR x = 0 OR x > 0);
```

The variant specification that uses an ELSE in place of the condition x > 0 generates a simpler disjointness TCC (just the first of the three conjuncts in sign_cond_TCC2), and no coverage TCC.

2.2 The PVS TABLE Construct

PVS has TABLE constructs that provide a fairly attractive input syntax for one- and two-dimensional tables and that are LaTeX-printed as true tables (the example *Parnas_Fig1* that appears later illustrates this). Their semantic treatment derives directly from the COND construct.

2.2.1 One-Dimensional Tables. The simplest tables in PVS are one-dimensional. In their *vertical* format, they simply replace the -> and , of COND cases by | and ||, respectively, and introduce each case with |; they also add a final || and change the keyword from COND to TABLE. The *sign* example is therefore transformed from a COND to the TABLE shown in 3. Note that the horizontal lines are simply comments (comments in PVS are introduced by %).

```
sign_vtable(x): signs = TABLE
            %-------------%
            | x < 0 | -1 ||
            %-------------%
            | x = 0 |  0 ||
            %-------------%
            | x > 0 |  1 ||
ENDTABLE    %-------------%
```

3

```
sign_htable(x): signs = TABLE
            %-------------------%
            |[ x<0 | x=0 | x>0 ]|
            %-------------------%
            |  -1  |  0  |  1  ||
ENDTABLE    %-------------------%
```

4

One-dimensional *horizontal* tables present the information in a different order, and use |[...]| to alert the parser to this fact, as illustrated in 4.

Both these tabular specifications are equivalent to sign_cond, generate exactly the same TCCs, and are treated the same in proofs. Notice that tables require no extensions to the PVS theorem prover, and the full repertoire of proof commands may be applied to constructions involving tables—for example, it is possible to rewrite with an expression whose right hand side is a table. Note, however, that PVS remembers the syntactic form used in a specification and always prints it out the same way it was typed in; thus, the prover will print a table as a table, even though it is treated semantically as a COND (which is itself treated as an IF-THEN-ELSE). Of course, the special syntactic treatment is lost once a proof step (e.g., one that "lifts" IF-THEN-ELSE constructs to the top level) has transformed the structures appearing in a sequent.

2.2.2 Blank Entries. Suppose we reformulated our *sign* example to take a natural number, rather than an integer, as its argument. The $x < 0$ case can no longer arise and can be omitted from the table. In some circumstances, however, we may wish to make it patently clear that this case should not occur and we can do this by including the case, but with a blank entry for the value of the expression.

```
sign_htable(x: nat): signs = TABLE %------------------%
                                |[ x<0 | x=0 | x>0 ]|
                                %------------------%
                                |      | 0  | 1  ||
                          ENDTABLE %------------------%
```

The presence of blank entries changes the coverage TCC: this must now ensure that the disjunction of all the conditions with non-blank entries is true. Notice this requires a TCC to be generated even when an **ELSE** case is present.

In one-dimensional tables, blank entries can always be removed by simply deleting the entire case; this is not so with two-dimensional tables, however, where the accessibility of an entry may depend on the conditions labeling *both* its row and column. We describe an example later.

2.2.3 Enumeration Tables. These are a syntactic variation that provide more succinct representation when the conditions to a table are all of the form $x =$ *expression* for some single identifier x. In an enumeration table, the identifier concerned follows the **TABLE** keyword, and the conditions of the table simply list the *expressions*; a two-dimensional example appears below in [5].

Enumeration tables are an important special case because their TCCs are often easily decidable, and this allows some important optimizations. Observe that the number of conjuncts in a disjointness TCC grows as the square of the number of conditions; when enumerating over the values of an enumeration type, it is not uncommon to have tens or hundreds of conditions, and thus thousands of conjuncts in the disjointness TCC. It is unwieldy and slow to display such massive TCCs to the user. PVS therefore recognizes this case and treats it specially: when the expressions in an enumeration table are all constructors of a single datatype (and the values of an enumeration type are exactly these), the disjointness and coverage conditions are trivially decidable and are checked internally by the typechecker, which also translates such tables into a datatype **CASES** expression, rather than a **COND**.[2] Another special case arises when the expressions of an enumeration table are all literal values of some type (the usual case is values from some range of integers); again, the disjointness TCC is easily decidable and can be checked internally by the typechecker (the coverage TCC can require theorem proving and is generated normally). A table is immediately flagged as illegal if such internal checks reveal a false TCC.

2.2.4 Two-Dimensional Tables. Two-dimensional tables are treated as nested **COND** (or **CASES**) constructs; more particularly, the columns are nested within the rows. Here is a trivial example of a two-dimensional enumeration table in which the rows enumerate the values of a type **state** and the columns enumerate the values of a type **input**.

[2] The prover can provide greater automation for the **CASES** expression. The user could use a **CASES** construct directly in the one-dimensional case; the main benefit in providing the translation automatically is with two-dimensional tables.

```
example(state,input): some_type = TABLE  state  ,  input
                                                %---------%
                                                |[ x | y |]
                                          %-----------------%
                                          |   a    | p | q ||
                                          %-----------------%
                                          |   b    | q | q ||
                                ENDTABLE  %-----------------%
```

5

This translates internally to the following.

```
COND
  state = a -> COND input = x -> p,  input = y -> q ENDCOND,
  state = b -> COND input = x -> q,  input = y -> q ENDCOND
ENDCOND
```

Notice that this translation causes disjointness and coverage TCCs for the columns to be generated several times—once for each row. For example, the coverage TCCs generated for the two inner CONDs above have the following form.

```
coverage_a: OBLIGATION state = a IMPLIES input = x OR input = y
coverage_a: OBLIGATION state = b IMPLIES input = x OR input = y
```

These appear redundant, so we might be tempted to use the following, apparently equivalent, translation.

```
LET  x1 = COND input = x -> p, input = y -> q ENDCOND,
     x2 = COND input = x -> q, input = y -> q ENDCOND
IN   COND state = a -> x1, state = b -> x2 ENDCOND
```

This generates the following single, simple coverage TCC for the columns.

```
coverage_TCC: OBLIGATION input = x OR input = y
```

The problem with this translation is that there may be subtype TCCs generated from the terms corresponding to p and q that must be conditioned on the expressions corresponding to a and b in order to be provable. Here is an example due to Parnas [15, Figure 1] that illustrates this. We exhibit this example in the form output by the PVS LaTeX-printer.

Parnas_Fig1$((y, x : \text{real})) : \text{real} =$

	$y = 27$	$y > 27$	$y < 27$
$x = 3$	$27 + \sqrt{27}$	$54 + \sqrt{27}$	$y^2 + 3$
$x < 3$	$27 + \sqrt{-(x-3)}$	$y + \sqrt{-(x-3)}$	$y^2 + (x-3)^2$
$x > 3$	$27 + \sqrt{x-3}$	$2 \times y + \sqrt{x-3}$	$y^2 + (3-x)^2$

The subtype constraint on the argument to the square root function (namely, that it be nonnegative) generates TCCs in the second and third rows that are true only when the corresponding row constraints are taken into account. The LET form translation loses this information. The advantage of the simple translation, which is the one used in PVS, is that it provides more precise (i.e., weaker but still adequate) TCCs, and therefore admits more specifications.

2.3 Applications

The PVS table constructs described above have been used in several applications performed by ourselves and others—indeed, some elements in the PVS treatment of tables (notably, blank entries, and the optimizations for enumeration tables) evolved in response to these applications.

In one application, PVS is being employed in analysis of new requirements documented in "Change Requests" (CRs) for the flight software of the Space Shuttle. This work is undertaken as part of a project involving staff from several NASA Centers (Langley, Johnson, and JPL) and Requirements Analysts (RAs) from the team at Lockheed Martin (formerly IBM) that develops this software. Running alongside what is generally considered an exemplary (though manual) process for requirements review, this experiment provides useful data on the effectiveness of automated formal analyses [2, 18].

One of the CRs focused on improving the display of flight information to Shuttle pilots guiding the critical initial bank onto the "Heading Alignment Cylinder" (HAC) during descent. The CR documented key portions of the required control logic in tabular form, and was readily formalized using PVS tables; a small representative example is reproduced in Appendix A. Attempts to discharge the TCCs generated by these tables immediately indicated the need to document implicit "domain knowledge," including constraints such as "`Major Mode = 305 or 603` implies `iphase` ≤ 3," and "`wowlon` can be true only if `Major Mode = 305 or 603`." Such domain knowledge was incorporated into the specification using dependent predicate subtyping and was gradually extended and refined through an iterative process that relied on the automated strategies for proving TCCs that are built in to PVS.

Observe that proofs of the HAC TCCs could be automated because necessary domain knowledge was supplied through the type system, using predicate and dependent subtyping. For example, the constraints mentioned above were specified as follows (`iphase` and `wowlon` are record fields; notice that the latter has a type that is a subtype of `bool`!).

```
iphase: {p: iphase | (mode = mm602 => p >= 4) AND
                     ((mode = mm305 OR mode = mm603) => p <= 3)}
wowlon: {b: bool | b => (mode = mm305 OR mode = mm603)}
```

The PVS prover can make very effective and automated use of information supplied in this way; a system lacking such a rich type system would probably require an interactive proof to provide the domain knowledge in the form of axioms. (Of course, PVS's decision procedures for linear arithmetic also contributed to the automation of these proofs.)

After incorporating all constraints identified by the RAs, it was found that the conditions for several rows in one table still overlapped, and this led to identification of a missing conjunct in some of the conditions. In addition to

discovery of this error, the requirements analysts felt that explicit identification and documentation of the domain knowledge was a valuable product of the analysis [18].

Another application for PVS tables has been in verification of fast hardware division algorithms. The notorious Pentium FDIV bug, which is reported to have cost Intel $475 million, was due to bad entries in the quotient lookup table for an SRT divider. Triangular-shaped regions at top and bottom of these tables are never referenced by the algorithm; the Pentium error was that certain entries believed to be in this inaccessible region, and containing arbitrary data, were, in fact, sometimes referenced during execution [16].

An SRT division algorithm similar to that used in the Pentium has been specified and verified in PVS [19]. The quotient lookup table for this algorithm was specified as a PVS table (reproduced in Appendix B) which uses blank entries to indicate those regions of the table that are believed to be inaccessible. PVS generates 23 coverage TCCs to ensure that these entries will never be encountered; verification of the algorithm (which can be done largely automatically in PVS) then ensures that all the nonblank table entries are correct. Injection of an error similar to that in the Pentium leads to a failed TCC proof whose final sequent is a counterexample that highlights the error [19]. Miner and Leathrum have used this capability of PVS to develop several new SRT tables [11], each in less than three hours.

3 Decision Tables

Decision tables associate Boolean expressions with the "decision" or output to be generated when a particular expression is true. There are many kinds of decision tables; the ones considered here are from a requirements engineering methodology developed for avionics systems by Lance Sherry of Honeywell [22], and given mechanized support in TableWise, developed by Hoover and Chen at ORA [8]. The following is a simple decision table (taken from [8, Table 2]).

	Operational Procedure					
Input Variables	Takeoff		Climb		Climb_Int_level	Cruise
Flightphase	climb	climb	climb	climb	climb	cruise
AC_Alt > 400	true	true	*	*	*	*
compare(AC_Alt, Acc_Alt)	LT	LT	GE	GE	*	GT
Alt_Capt_Hold	false	true	false	true	true	true
compare(Alt_Target, prev_Alt_Target)	*	GT	*	GT	*	EQ

This table describes the conditions under which each of the four "operational procedures" **Takeoff**, **Climb**, **Climb_Int_Level**, and **Cruise** should be selected. Each of the columns beneath the name of an operational procedure gives a conjunction of conditions under which that procedure should be selected

(where * indicates "don't care"). For example, the third and fourth columns in the body of the table indicate that the operational procedure `Climb` should be used if the `Flightphase` is `climb`, `AC_Alt` is greater than or equal to `Acc_Alt`, and either `Alt_Capt_Hold` is *false*, or it is *true* and `Alt_Target` is greater than `prev_Alt_Target`. The columns forming a subtable beneath each operational procedure are similar to the AND/OR tables used in the RSML notation of Leveson and colleagues [10].

The PVS `TABLE` construct cannot represent this type of decision table directly: we need some additional mechanism to represent a conjunction such as

$$(\texttt{Flightphase} = \texttt{climb}) \wedge (\texttt{AC\_Alt} \geq \texttt{Acc\_Alt}) \wedge \neg\texttt{Alt\_Capt\_Hold}$$

by the compact list given in the third column of the table.

Now the list `(climb, *, GE, false, *)` from that column can be interpreted as the argument list to a function `X` that treats the first element as a function to be applied to `Flightphase`, the second as a function to be applied to the expression `AC_Alt > 400` and so on, as follows.

```
X(a,b,c,d,e): bool =
    a(Flightphase) & b(AC_Alt > 400) & c(AC_Alt,Acc_Alt)
       & d(Alt_Capt_Hold) & e(Alt_Target,prev_Alt_Target)
```

We can then use this construction to specify the third column of the decision table as the following row from a vertical one-dimensional PVS table; the complete table is shown in Appendix C (taken from [12], where full details may be found).

```
%----------|-------|-------|-------|-------|-----------------%
| X(climb? ,   *    ,  GE   , false ,   *  )| Climb           ||
%----------|-------|-------|-------|-------|-----------------%
```

The functions appearing in the argument list to `X` are defined as follows (note that * is overloaded and that `climb?` is a recognizer for an enumerated type).

```
q: VAR bool                          x, y: VAR nat
false(q): bool = NOT q               GE(x, y): bool = x >= y
    *(q): bool = TRUE                 *(x, y): bool = TRUE
```

The disjointness TCC from this table immediately identifies two overlapping cases, while the coverage TCC identifies four that are missing. For example, one of the four unproved sequents[3] from the coverage TCC is the following.

[3] PVS uses a sequent calculus presentation whose interpretation is that the conjunction of formulas above the turnstile line (|------) should imply the disjunction of formulas below the line. The appearance of a formula on one side of the line is equivalent to its negation on the other, and this structural rule is used to eliminate top-level negations. Names with embedded ! characters are Skolem constants derived from variables with the same root name.

```
decision_table_TCC2.1 :                                              6

  |-------
[1]   AC_Alt!1 > 400
[2]   Alt_Capt_Hold!1
[3]   AC_Alt!1 >= Acc_Alt!1
```

Unproven sequents such as this, with no formulas above the line, indicate the failure to select an operational procedure when all the formulas below the line are false. This one, for example, identifies the failure to consider the case when `AC_Alt` is not greater than `400`, `Alt_Capt_Hold` is *false*, and `AC_Alt` is less than `Acc_Alt`. The six flaws identified in this way are identical to those found in this example by the special-purpose tool TableWise [8].

Unlike PVS, TableWise presents the anomalies that it discovers in a tabular form similar to that of the original decision table; TableWise can also generate executable Ada code and English language documentation from decision tables. These benefits are representative of those that can be achieved with a special-purpose tool. On the other hand, PVS's more powerful deductive capabilities also provide benefits. For example, PVS can settle disjointness and coverage TCCs that depend on properties more general than the simple Boolean and arithmetic relations built in to TableWise and similar tools. The limitations of these tools are illustrated by Heimdahl [3], who describes spurious error reports when a completeness and consistency checking tool for the AND/OR tables of RSML (developed with Leveson [5]) was applied to TCAS II. These spurious reports were due to the presence of arithmetic and defined functions whose properties are beyond the reach of the BDD-based tautology checker incorporated in the tool. As Heimdahl notes [3, page 81], a theorem prover is needed to settle such properties; he and Czerny are now experimenting with PVS for this purpose [4].

A theorem prover such as PVS can also examine questions beyond simple completeness and consistency. For example, the incompleteness and inconsistencies detected in the example decision table can be remedied by adding an `ELSE` clause and by replacing the second and third "don't care" entries under `Climb_Int_level` by `false` and `LT`, respectively. The TCC generated by this modified specification is proved automatically by PVS, so we may proceed to examine general properties of the decision table. To check that the specification matches our intent, we can use conjectures that we believe to be true as "challenges." For example, we may believe that when `AC_Alt = Acc_Alt`, the operational procedure selected should match the `Flightphase`. We can check this in the case that the `Flightphase` is `cruise` using the following challenge.

```
test: THEOREM AC_Alt = Acc_Alt =>
  decision_table(cruise, AC_Alt, Acc_Alt,
                 Alt_Target, prev_Alt_Target, Alt_Capt_Hold) = Cruise
```

This is easily proved by PVS's standard `(grind)` strategy. However, when we try the corresponding challenge for the case where `Flightphase` is `climb`, we

discover that the conjecture is not proved, and actually is false in the case where `Alt_Capt_Hold` is *true* and `Alt_Target <= prev_Alt_Target`, thereby exposing a flaw in either our expectations or our formalization of the specification. Mechanically supported challenges of this kind illustrate the utility of undertaking the analysis of tabular specifications in a context that provides theorem proving. Special-purpose tools for tabular specifications generally provide only completeness and consistency checking, and perhaps some form of simulation. Such tools would help identify the anomaly just described only if we happened to choose to simulate a case where `Alt_Capt_Hold` is *true* and `Alt_Target <= prev_Alt_Target`.

4 Transition Relations and Model Checking

Decision tables provide a way to specify the selection of operational procedures to be executed at each step. However, the model of computation that repeatedly performs these selection and execution steps is understood informally and is not explicit in the PVS specifications. Consequently, it is not possible to pose and examine overall system properties—such as whether a certain property is invariant, or another is reachable—without formalizing more of the underlying model of computation. *Transition relations* provide a way to do this, and the SCR method is a way to present such relations in a tabular manner [7].

The following is a typical SCR "mode transition table" (taken from Atlee and Gannon [1, Table 2]). This system, a simplified automobile cruise control, has four modes (`off`, `inactive`, `cruise`, and `override`) and the table describes the conditions under which it makes transitions from one mode to another.

Current Mode	Conditions							Next Mode
	Ignited	Running	Toofast	Brake	Activate	Deactivate	Resume	
Off	@T	-	-	-	-	-	-	Inactive
Inactive	@F	-	-	-	-	-	-	Off
	T	T	-	F	@T	-	-	Cruise
Cruise	@F	-	-	-	-	-	-	Off
	-	@F	-	-	-	-	-	Inactive
	-	-	@T	-	-	-	-	Inactive
	-	-	-	@T	-	-	-	Override
	-	-	-	-	-	@T	-	Override
Override	@F	-	-	-	-	-	-	Off
	-	@F	-	-	-	-	-	Inactive
	T	T	-	F	@T	-	-	Cruise
	T	T	-	F	-	-	@T	Cruise

An `@T` entry indicates the case where the condition labeling that column changes from *false* to *true*, while `@F` indicates the opposite transition; a `T` entry indicates the case where the condition labeling that column remains *true* through the transition, `F` indicates the case where it remains *false*, and a dash indicates

"don't care." Thus the third row indicates that the system transitions from the **Inactive** mode to the **Cruise** mode if **Activate** goes *true*, while **Ignited** and **Running** remain *true* and **Brake** remains *false*.

To model this type of specification in PVS, we specify a **condition** as a predicate on inputs to the system, then **atT** (which represents **@T**) is a higher order function that takes a condition and returns a relation on pairs of inputs (namely, one that is *true* when the condition is *false* when applied to the first and *true* when applied to the second). The constructions for **atF** (representing **@F**), **T**, **F**, and **dc** (representing "don't care") are specified similarly.

```
scr[ input, mode, output: TYPE ]: THEORY
BEGIN
  condition: TYPE = pred[input]
  p,q: VAR input
  P: VAR condition

  atT(P)(p,q): bool = NOT P(p) & P(q)       % @T(P)
  atF(P)(p,q): bool = P(p) & NOT P(q)       % @F(P)
  T(P)(p,q):   bool = P(p) & P(q)
  F(P)(p,q):   bool = NOT P(p) & NOT P(q)
  dc(P)(p,q):  bool = true                  % don't care
  ...
```

With these constructions, the mode transition table shown earlier can be represented in PVS as follows (for brevity, we show only the transitions from the **Inactive** mode, corresponding to the second and third rows of the table; the complete table is shown in Appendix D, and full details are given in [12]).

```
event_constructor: TYPE = [condition -> event]
EC: TYPE = event_constructor
PC(A,B,C,D,E,F,G)(a,b,c,d,e,f,g)(p,q):bool = A(a)(p,q) & B(b)(p,q)
     & C(c)(p,q) & D(d)(p,q) & E(e)(p,q) & F(f)(p,q) & G(g)(p,q)
% Note: PC stands for "pairwise conjunction"

original(s: modes, (p, q: monitored_vars)): modes =
LET
 x = (ignited, running, toofast, brake, activate, deactivate, resume),
 X = (LAMBDA (a,b,c,d,e,f,g:EC): PC(a,b,c,d,e,f,g)(x)(p,q))
IN TABLE s
  ...
  |inactive| TABLE %----|----|----|----|----|----|----|-----|----------||
                  |X(   atF , dc , dc , dc , dc , dc , dc )| off      ||
                  %----|----|----|----|----|----|----|-----|----------||
                  |X(    T  , T  , dc , F  ,atT , dc , dc )| cruise   ||
                  %----|----|----|----|----|----|----|-----|----------||
                  |    ELSE                                 | inactive ||
  ENDTABLE ||     %----|-----------------------------------|----------||
  ...
```

Typechecking this specification generates several TCCs; those for the transitions from mode **inactive** are proved automatically, but those from modes **cruise** and **override** are not. These unproved TCCs yield subgoals that pinpoint problems in the specification, rather in the way that [6] identified problems in the decision table. For example, the successor to **cruise** mode is ambiguous in the case where **toofast** and **deactivate** both go from *false* to *true*: the first of these causes a transition to **inactive** mode, while the second causes a transition to **override** mode. Repairing these flaws requires several changes to the table and—as with the Space Shuttle example—adding some "domain knowledge" (such as that **toofast** implies **running**).

Because a mode transition table specifies how the system proceeds from one mode to another, we can examine properties of the computations that this induces. To do this, we first need to derive the transition relation on states that is implicit in a mode table. We identify the instantaneous **state** of the system with its current mode and the current values of its input variables. We specify this as a record in PVS; a transition relation is a predicate on pairs of such states.

```
state: TYPE = [# mode: mode, vars: input #]
transition_relation: TYPE = pred[[state, state]]
```

Recall that a mode transition table has the following signature.

```
mode_table: TYPE = [mode, input, input -> mode]
```

We can therefore define a function **trans** that takes a mode table and returns the corresponding state transition relation.

```
trans(mt: mode_table): transition_relation =
  (LAMBDA (s,t: state): mode(t) = mt(mode(s), vars(s), vars(t)))
```

The branching time temporal logic CTL provides a convenient way to specify certain properties of the computations induced by a transition relation, and PVS can automatically verify CTL formulas for transition relations over finite types by using a decision procedure for Park's μ-calculus to provide CTL model checking [17]. An example of a property about this specification that can be specified in CTL is the following invariant.

> In **cruise** mode, the engine is **running**, the vehicle is not going **toofast**, the **brake** is not on, and **deactivate** is not selected.

We can examine this property with PVS in the following manner.

```
IMPORTING MU@ctlops, cruise_tab
p,q,r: var state
trans: transition_relation = trans(deterministic)
init(p): bool = off?(p) & NOT ignited(p)

safe4: THEOREM init(p) => AG(trans,(LAMBDA q:
  cruise?(q)
    => running(q) & NOT (toofast(q) OR brake(q) OR deactivate?(q))))(p)

safe5: THEOREM init(p)
         => AG(trans, (LAMBDA q: override?(q) => running(q)))(p)
```

Here, **cruise_tab** is the PVS theory that defines the mode table **deterministic** (formed by correcting the errors found in the table **original** discussed above), and **ctlops** is the PVS theory (from the library **MU**) that defines the CTL operators. The function **trans** introduced above is applied to the mode table **deterministic** to construct a transition relation (also called **trans**). We characterize the initial state as one whose mode is **off** and in which the engine is not **ignited**, and specify (as **safe4**) the invariant mentioned above (**AG** is the CTL operator meaning "in every reachable state"). Another plausible invariant property is specified by the formula **safe5**. The PVS **model-check** command verifies formula **safe5** but fails on **safe4**. This prompts closer examination of the specification and reveals that, although **cruise** mode is exited when **toofast** goes *true*, the transitions into **cruise** mode neglect to check that **toofast** is *false* before making the transition. The correction is to add the condition **F(toofast)** to the three transitions into **cruise** mode, and PVS is able to verify the formula **safe4** for the corrected specification.

Similar to the TableWise tool for decision tables, Heitmeyer and colleagues have developed the SCR* tool for checking consistency of SCR tabular specifications [6], while Atlee and colleagues have developed a translator that turns SCR tables into a form acceptable to the SMV model checker [23]. These special-purpose tools have the advantage of being closely tailored to their intended uses and are scalable to larger examples than is convenient for the PVS treatment. On the other hand, the PVS treatment required no customized development: it simply builds on capabilities such as tables, higher-order logic, theorem proving, and model checking that are already present in PVS.

Furthermore, the PVS treatment can draw on the full resources of the language and system to combine methods in novel ways, or to conduct customized analyses. For example, we have used a variant of PVS's treatment of SCR tables to specify the nondeterministic mode transitions of interacting "climb" and "level" components in the requirements for a simple "autopilot" [12, section 4.3]. The transitions of the components were specified as separate tables and combined by disjunction (representing interleaving concurrency). The combined specification was then tested against a number of challenge properties using model checking. A deterministic "implementation" specification of the autopilot was constructed from two "phases" using relational composition to specify

sequential execution. This specification was also tested against the challenge properties using model checking. Finally, model checking was used to show that the behaviors induced by the requirements and the implementation specifications are equivalent (this property can be expressed as a CTL formula).

5 Conclusion

We have described PVS's capabilities for representing tabular specifications, illustrated how these interact synergistically with other capabilities such as typechecker-generated proof obligations, dependent typing, higher-order functions, model checking, and general theorem proving, and described some applications. We demonstrated how these capabilities of the PVS language and verification system can be used in combination to provide customized support for existing methodologies for documenting and analyzing requirements. Because they use only the standard capabilities of PVS, users can adapt and extend these customizations to suit their own needs.

The generic support provided for tables and for model checking in PVS may be compared with the more specialized support provided in tools such as ORA's TableWise [8], NRL's SCR* [6,7], and Leveson and Heimdahl's consistency checker for RSML [5]. Dedicated, lightweight tools such as these are likely to be superior to a heavyweight, generic system such as PVS for their chosen purposes. Our goal in applying PVS to these problems is not to compete with specialized tools but to complement them. The generic capabilities of PVS can be used to prototype some of the capabilities of specialized tools (this was done in the development of TableWise), and can also be used to supplement their capabilities when comprehensive theorem proving and model checking power is needed.

Acknowledgments

Examples undertaken by Ricky Butler, Ben Di Vito, and Paul Miner of NASA Langley Research Center, Steve Miller of Collins Commercial Avionics and Harald Rueß of Universität Ulm, and suggestions by Connie Heitmeyer of the Naval Research Laboratory, were instrumental in shaping the PVS table constructs. Comments by the anonymous referees improved the presentation of this paper.

References

Papers by SRI authors are generally available from `http://www.csl.sri.com/fm.html`.

1. Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. In *SIGSOFT '91: Software for Critical Systems*, pages 16–28, New Orleans, LA, December 1991. Published as ACM SIGSOFT Engineering Notes, Volume 16, Number 5.

2. Judith Crow and Ben L. Di Vito. Formalizing space shuttle software requirements: Four case studies. Submitted for publication, 1997.
3. Mats P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In Steven J. Zeil, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 79–83, San Diego, CA, January 1996. Association for Computing Machinery.
4. Mats P. E. Heimdahl and Barbara J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *IEEE High-Assurance Systems Engineering Workshop (HASE '96)*, pages 252–262, Niagara on the Lake, Canada, October 1996.
5. Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency analysis of state-based requirements. In *17th International Conference on Software Engineering*, pages 3–14, Seattle, WA, April 1995. IEEE Computer Society.
6. Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In COMPASS [9], pages 109–122.
7. Constance Heitmeyer, Bruce Labaw, and Daniel Kiskis. Consistency checking of SCR-style requirements specifications. In *International Symposium on Requirements Engineering*, York, England, March 1995. IEEE Computer Society.
8. D. N. Hoover and Zewei Chen. Tablewise, a decision table tool. In COMPASS [9], pages 97–108.
9. *COMPASS '95 (Proceedings of the Tenth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1995. IEEE Washington Section.
10. Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.
11. Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 64–78, Palo Alto, CA, November 1996. Springer-Verlag.
12. Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Available, with specification files, at `http://www.csl.sri.com/csl-95-12.html`.
13. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
14. Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.
15. David Lorge Parnas. Tabular representation of relations. Technical Report CRL Report 260, Telecommunications Research Institute of Ontario (TRIO), Faculty of Engineering, McMaster University, Hamilton, Ontario, Canada, October 1992.
16. Vaughan Pratt. Anatomy of the Pentium bug. In *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107, Aarhus, Denmark, May 1995. Springer-Verlag.
17. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification,*

CAV '95, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

18. Larry W. Roberts and Mike Beims. Using formal methods to assist in the requirements analysis of the Space Shuttle HAC Change Request (CR 90960E). Technical Report JSC-27599, NASA Johnson Space Center, Houston, TX, September 1996.
19. H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 123–134, New Brunswick, NJ, July/August 1996. Springer-Verlag.
20. John Rushby. Mechanizing formal methods: Opportunities and challenges. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 105–113, Limerick, Ireland, September 1995. Springer-Verlag.
21. Natarajan Shankar. Unifying verification paradigms. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*, pages 22–39, Uppsala, Sweden, September 1996. Springer-Verlag.
22. Lance Sherry. A structured approach to requirements specification for software-based systems using operational procedures. In *13th AIAA/IEEE Digital Avionics Systems Conference*, pages 64–69, Phoenix, AZ, October 1994.
23. Tirumale Sreemani and Joanne M. Atlee. Feasibility of model checking software requirements. In *COMPASS '96 (Proceedings of the Eleventh Annual Conference on Computer Assurance)*, pages 77–88, Gaithersburg, MD, June 1996. IEEE Washington Section.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Appendix

A HAC Requirements Table Expressed in PVS

```
switch_position: TYPE = {low, medium, high}
major_mode:      TYPE = {mm301, mm302, mm303, mm304, mm305, mm602, mm603}
iphase:          TYPE = {n: nat | n <= 6} CONTAINING 0

ADI_error_inputs: TYPE =
    [# mode: major_mode,
       switch_position: switch_position,
       iphase: {p: iphase | (mode = mm602 => p >= 4) AND
                            ((mode = mm305 OR mode = mm603) => p <= 3)},
       wowlon: {b: bool | b => (mode = mm305 OR mode = mm603)} #]

ADI_error_scale_deflection(A: ADI_error_inputs) : [real, real, real] =
 LET mode = mode(A), switch_position = switch_position(A),
    iphase = iphase(A), wowlon = wowlon(A) IN
 TABLE  %  Result is of form: [roll error, pitch error, yaw error]
                    ,                    switch_position
                    %-------------------------------------------------------%
                    |[     high      |     medium     |      low       ]|
 %-------------------------------------------------------------------------%
 | mode = mm301 OR
   mode = mm302 OR
   mode = mm303     | (10, 10, 10)   | (5, 5, 5)      | (1, 1, 1)      ||
 %-------------------------------------------------------------------------%
 | mode = mm304 OR
  (mode = mm602 AND
    (iphase = 4 OR
     iphase = 6))   | (25, 5, 5/2)   | (25, 2, 5/2)   | (10, 1, 5/2)   ||
 %-------------------------------------------------------------------------%
 | mode = mm602 AND
   iphase = 5       | (25, 5/4, 5/2) | (25, 5/4, 5/2) | (10, 1/2, 5/2) ||
 %-------------------------------------------------------------------------%
 | (mode = mm305 OR
   mode = mm603) AND
    NOT wowlon      | (25, 5/4, 5/2) | (25, 5/4, 5/2) | (10, 1/2, 5/2) ||
 %-------------------------------------------------------------------------%
 | wowlon           | (20, 10, 5/2)  | (5, 5, 5/2)    | (1, 1, 5/2)    ||
 %-------------------------------------------------------------------------%
ENDTABLE
```

B Quotient Lookup Table for SRT Divider

```
q(D: bvec[3], (P: bvec[7] | estimation_bound?(valD(D), valP(P)))):
     subrange(-2, 2) =
LET a = -(2 - P(1) * P(0)),
    b = -(2 - P(1)),
    c = 1 + P(1),
    d = -(1 - P(1)),
    e = P(1),
    Dp:nat = bv2pattern(D),
    Ptruncp:nat = bv2pattern(P^(6,2))
IN TABLE  Ptruncp,                  Dp
                |[ 000| 001| 010| 011| 100| 101| 110| 111]|
            %-----------------------------------------------%
            |01010|    |    |    |    |    |    |    |  2 ||
            |01001|    |    |    |    |    |  2 |  2 |  2 ||
            |01000|    |    |    |    |  2 |  2 |  2 |  2 ||
            |00111|    |    |  2 |  2 |  2 |  2 |  2 |  2 ||
            |00110|    |  2 |  2 |  2 |  2 |  2 |  2 |  2 ||
            |00101|  2 |  2 |  2 |  2 |  2 |  2 |  2 |  1 ||
            |00100|  2 |  2 |  2 |  2 |  c |  1 |  1 |  1 ||
            |00011|  2 |  c |  1 |  1 |  1 |  1 |  1 |  1 ||
            |00010|  1 |  1 |  1 |  1 |  1 |  1 |  1 |  1 ||
            |00001|  1 |  1 |  1 |  1 |  e |  0 |  0 |  0 ||
            |00000|  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 ||
            |11111|  0 |  0 |  0 |  0 |  0 |  0 |  0 |  0 ||
            |11110| -1 | -1 |  d |  d |  0 |  0 |  0 |  0 ||
            |11101| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 ||
            |11100|  a |  b | -1 | -1 | -1 | -1 | -1 | -1 ||
            |11011| -2 | -2 | -2 |  b | -1 | -1 | -1 | -1 ||
            |11010| -2 | -2 | -2 | -2 | -2 | -2 |  b | -1 ||
            |11001| -2 | -2 | -2 | -2 | -2 | -2 | -2 | -2 ||
            |11000|    |    | -2 | -2 | -2 | -2 | -2 | -2 ||
            |10111|    |    |    | -2 | -2 | -2 | -2 | -2 ||
            |10110|    |    |    |    |    | -2 | -2 | -2 ||
            |10101|    |    |    |    |    |    | -2 | -2 ||
            %-----------------------------------------------%
    ENDTABLE
```

C Example Decision Table

```
q:VAR bool

 true(q): bool = q
false(q): bool = NOT q
    *(q): bool = TRUE

x,y:VAR nat

GT(x, y): bool = x > y          LT(x, y): bool = x < y
GE(x, y): bool = x >= y         LE(x, y): bool = x <= y ;
EQ(x, y): bool = x = y           *(x, y): bool = TRUE

operational_procedures: TYPE = {Takeoff, Climb, Climb_Int_Level, Cruise}

flight_phases: TYPE = {climb, cruise}

Flightphase: VAR flight_phases
AC_Alt, Acc_Alt, Alt_Target, prev_Alt_Target: VAR nat
Alt_Capt_Hold: VAR bool

decision_table(Flightphase, AC_Alt, Acc_Alt, Alt_Target,
               Prev_Alt_Target, Alt_Capt_Hold): operational_procedures=
LET X = (LAMBDA (a: pred[flight_phases]), (b: pred[bool]),
            (c: pred[[nat,nat]]), (d: pred[bool]), (e: pred[[nat,nat]]):
   a(Flightphase) &
           b(AC_Alt > 400) &
                   c(AC_Alt,Acc_Alt) &
                          d(Alt_Capt_Hold) &
                                 e(Alt_Target,prev_Alt_Target)) IN TABLE
%       |        |       |       |       |
%       |        |       |       |       |
%       v        v       v       v       v   Operational Procedure
  %-----------|-------|-------|-------|-------|------------- ----%
  | X(climb? , true  ,   LT  , false ,   * ) | Takeoff         ||
  %-----------|-------|-------|-------|-------|-----------------%
  | X(climb? , true  ,   LT  , true  ,   GT) | Takeoff         ||
  %-----------|-------|-------|-------|-------|-----------------%
  | X(climb? ,   *   ,   GE  , false ,   * ) | Climb           ||
  %-----------|-------|-------|-------|-------|-----------------%
  | X(climb? ,   *   ,   GE  , true  ,   GT) | Climb           ||
  %-----------|-------|-------|-------|-------|-----------------%
  | X(climb? ,   *   ,   *   , true  ,   * ) | Climb_Int_Level ||
  %-----------|-------|-------|-------|-------|-----------------%
  | X(cruise?,   *   ,   GT  , true  ,   EQ) | Cruise          ||
  %-----------|-------|-------|-------|-------|-----------------%
ENDTABLE
```

D Example SCR Table

```
  event_constructor: TYPE = [condition -> event]
  EC: TYPE = event_constructor
  PC(A,B,C,D,E,F,G)(a,b,c,d,e,f,g)(p,q):bool = A(a)(p,q) & B(b)(p,q)
       & C(c)(p,q) & D(d)(p,q) & E(e)(p,q) & F(f)(p,q) & G(g)(p,q)
% Note: PC stands for "pairwise conjunction"

  original(s: modes, (p, q: monitored_vars)): modes =
  LET
    x: conds7 = (ignited, running, toofast, brake, activate, deactivate, resume),
    X = (LAMBDA (a,b,c,d,e,f,g:EC): PC(a,b,c,d,e,f,g)(x)(p,q))
  IN TABLE s
   |off| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atT , dc , dc , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                                | off      ||
      %----|----------------------------------|----------||
    ENDTABLE ||

   |inactive| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF , dc , dc , dc , dc , dc , dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    T  , T  , dc , F  ,atT , dc , dc )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                                | inactive ||
      %----|----------------------------------|----------||
     ENDTABLE ||

   |cruise| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF,  dc,  dc,  dc,  dc,  dc,  dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc ,atF , dc , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , dc ,atT , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , dc , dc ,atT , dc , dc , dc )| override ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc , dc , dc , dc , dc ,atT , dc )| override ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                                | cruise   ||
      %----|----------------------------------|----------||
     ENDTABLE ||

    |override| TABLE
      %----|----|----|----|----|----|----|-----|----------||
      |X(   atF , dc   dc , dc , dc , dc , dc )| off      ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    dc ,atF , dc , dc , dc , dc , dc )| inactive ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    T  , T  , dc , F  ,atT , dc , dc )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |X(    T  , T  , dc , F  , dc , dc ,atT )| cruise   ||
      %----|----|----|----|----|----|----|-----|----------||
      |    ELSE                                | override ||
      % ---|----------------------------------|----------||
     ENDTABLE ||
ENDTABLE
```

This article was processed using the LaTeX macro package with LLNCS style

Appears in *Applied Formal Methods—FM-Trends 98*, Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman Eds., Springer Verlag Lecture Notes in Computer Science Vol. 1641, pp. 338–345, Boppard, Germany, October 1998.

PVS: An Experience Report*

S. Owre, J. M. Rushby, N. Shankar, and D. W. J. Stringer-Calvert

Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA
{owre, rushby, shankar, dave_sc}@csl.sri.com
URL: http://pvs.csl.sri.com

Abstract. PVS is a comprehensive interactive tool for specification and verification combining an expressive specification language with an integrated suite of tools for theorem proving and model checking. PVS has many academic and industrial users and has been applied to a wide range of verification tasks. In this note, we summarize some of its applications.

1 Introduction to PVS

PVS (Prototype Verification System) is an environment for constructing clear and precise specifications and for efficient mechanized verification. It is designed to exploit the synergies between language and deduction, automation and interaction, and theorem proving and model checking. The PVS specification language is a typed higher-order logic with a richly expressive type system with predicate subtypes and dependent types. Typechecking in this language requires the services of a theorem prover to discharge proof obligations corresponding to subtyping constraints.

The development of PVS began in 1990, and it was first made publicly available in 1993. Subsequent releases have increased its robustness and speed, and added a host of new capabilities. The essential features of PVS have already been described in prior publications [30, 32, 40], and comprehensive details can be found in the system manuals that are available from the PVS web site at `http://pvs.csl.sri.com`. In this note, we indicate the capabilities of the system through a survey of some of the applications for which it has been used. Due to space constraints, this is only a small sampling of the applications that have been performed using PVS, and even those that are mentioned are often given without full citations (we generally cite only the most accessible and the most recent works). We apologize to all PVS users whose work is omitted or mentioned without citation, and refer all readers to the online PVS Bibliography for a comprehensive list of citations to work concerning PVS [38].

We divide PVS activities and applications into a few broad subject areas: library development, requirements analysis, hardware verification, fault-tolerant algorithms, distributed algorithms, semantic embeddings/backend support, real-time and hybrid systems, security and safety, and compiler correctness.

* The development of PVS was funded by SRI International through Internal R&D funds. Various applications and customizations have been funded by NSF Grants CCR-930044 and CCR-9509931, and by contracts F49620-95-C0044 from AFOSR, NAS1-20334 from NASA, and N00015-92-C-2177 from NRL.

2 PVS Library Development

A major cost in undertaking formal specification and verification is that of developing formalizations for all the "background knowledge" that is required. PVS libraries help reduce this cost by providing formalizations for many common mathematical domains. Good libraries are challenging to develop: not only must they provide foundational definitions and axiomatizations that are correct, together with a body of derived constructions and lemmata that are rich enough to support development of clean, succinct, and readable specifications, but they must express these in a way that allows the PVS theorem prover to make effective use of them.

The "prelude" library built in to PVS provides many useful definitions and theorems covering basic mathematical concepts such as sets, bags, functions, relations, and orderings, together with properties of real and integer arithmetic outside the domain of the PVS decision procedures (principally those involving nonlinear arithmetic).

External PVS libraries provide finite sets, floor and div/mod, bitvectors, coalgebras, real analysis, graphs, quaternions, μ-calculus, and linear and branching time temporal logics. Development of libraries is very much a community effort in which sharing, modification, and extension has allowed the PVS libraries to grow into effective, robust and reusable assets. For example, the library for undirected graphs was developed by NASA Langley to support a proof of Menger's theorem [7]. This was extended to directed graphs by the University of Utah to support analysis of PCI bus transactions [28], and subsequently re-adopted and generalized by NASA.

3 Requirements

There is extensive evidence that requirements capture is the most error-prone stage in the software engineering lifecycle, and that detection and removal of those errors at later stages is very costly. Requirements provide a fruitful application area for formal methods because relatively "lightweight" techniques have proved effective in detecting numerous and serious errors. PVS supports these activities by providing direct support for consistency and completeness checking of tabular specifications [31], and through the process of "formal challenges" [39] where expected properties are stated of a specification and examined by theorem proving or model checking.

PVS has been used by multiple NASA centers to analyze requirements for the Cassini Spacecraft [13] and for the Space Shuttle [9], and by the SafeFM project (University of London) in the analysis of requirements for avionics control systems [12].

4 Hardware Verification

Applications of PVS to hardware verification fall into two broad classes. One class is concerned with verification of the complete microarchitecture against

the instruction set architecture seen by machine code programmers. While the presence of pipelining and other optimizations introduces complexities, the basic approach to this class of verifications depends on efficient symbolic simulation and equality reasoning, which in PVS are achieved by its tight integration of cooperating decision procedures with rewriting, combined with BDD-based simplification. PVS has been used for the full or partial verification of microcoded avionics and Java processors developed by Rockwell Collins [18], as well as for a number of smaller DLX-like processors with complex pipelines.

The other class of hardware applications concerns the complex circuits, algorithms, and protocols that are the building blocks of modern processors; these applications are sufficiently difficult that success depends on finding an effective methodology. Examples include verification of SRT dividers and other arithmetic circuits at NASA [27] and SRI, out-of-order execution at the University of Utah and SRI [23] and the Weizmann Institute [36], and cache-coherence at Stanford University [33]. Some applications are best handled using a combination of tools; PVS was used in this way by Fujitsu for the validation of the high-level design of an ATM switch [37].

5 Fault-Tolerant Algorithms

Mechanisms for fault tolerance are a significant component of many safety-critical systems: they can account for half the software in a typical flight-control system, and are sufficiently complicated that they can become its primary source of failure! Verifications of practical fault-tolerant designs are quite difficult and are often achieved incrementally, as more real-world complexities are layered on to a basic algorithm. The parameterized theories and strict dependency checking of PVS help in these incremental constructions.

For example, formal analysis of Byzantine fault tolerant clock synchronization has been elaborated over nearly a decade, with contributions from SRI and NASA Langley (using a predecessor to PVS) and the University of Ulm, culminating in verification of the algorithm used in a commercial system for safety-critical automobile control [35]. Comparable developments at SRI, NASA, Allied Signal, and Ulm have verified practical algorithms for consensus, diagnosis, and group membership, together with overall architectures for state machine replication and time-triggered execution of synchronous algorithms.

6 Distributed Algorithms

The fault tolerance applications described above employ synchronous algorithms. Other distributed algorithms are often asynchronous and are generally modeled as transition relations. Safety properties are traditionally verified by invariance arguments, and generation of suitably strong invariants is the major methodological challenge. More recent approaches employ abstraction to a finite-state (or other tractable) system that can be model checked. PVS has a model checker integrated with its theorem prover, so that it is able to perform all the stages of

such approaches. Examples include communications protocols [19] and garbage collection algorithms, parallel simulation algorithms [44] and parallelizing techniques [8], and operating system buffer-cache management [34].

Current research focusses on methods for automating the generation of abstractions and invariants [1, 5, 41].

7 Semantic Embeddings and Backend Support

For some applications it is convenient to use a customized logic for both specification and reasoning. Such logics can be encoded in the higher-order logic of PVS using either shallow or deep semantic embeddings. Examples include the Duration Calculus [42], DisCo [26], the B method [29], and coalgebraic treatments of Java classes [25]. An advantage of these embeddings over dedicated verification support is that the full expressiveness and power of PVS is available for all the auxiliary concepts and data types that are required.

An API for semantic embeddings of other logics is currently under development; this will allow specifications and proofs to be presented directly in the notation of the embedded logic.

An alternative to semantic embedding is to use PVS to discharge proof obligations generated by the support tool for another language. This route has been explored at Michigan State [20] and Bremen [6] universities.

8 Real-Time and Hybrid Systems

Formal treatments of real-time systems often employ special temporal or Hoare logics. Some of these have been supported by semantic embedding in PVS, as described above; others include timed automata [4], the language Trio [2], and the compositional method of Hooman [22]. Applications include several standard test-pieces, such as the Fisher's mutual exclusion algorithm, the Generalized Railroad Crossing, and the Steam Boiler, as well as some realistic protocols.

A real-time kernel for supporting Ada95 applications on a uniprocessor embedded system has also been developed in PVS at the University of York [14].

9 Security and Safety

Strong protection of data belonging to different processes is required for both security and safety in several applications. A formulation of this property in terms of "noninterference" forms one of the PVS tutorial examples. More elaborate and realistic treatments based on the same idea have been developed for security at Secure Computing Corporation [21], and for safe "partitioning" in avionics at NASA [45] and Rockwell Collins [49].

Ongoing work at SRI is developing an efficient approach for the verification of cryptographic protocols, while the special security problems arising in active networks have been formalized at the University of Cincinnati [11].

10 Compiler Correctness

In most system developments, correctness of the translation from source code to object code is not a source of major concern. Testing is performed on object code, which is fortuitously effective in finding errors introduced during compilation, assembly and linking. For critical developments, however, further assurance may be required.

PVS has been used to perform the verification of a compiler for a small safety critical language [43], and to reason about object code in terms of flow graphs [47]. The Verifix project (`http://i44s11.info.uni-karlsruhe.de/~verifix/`) at the Universities of Karlsruhe, Kiel, and Ulm has verified several compilation and optimization algorithms (including some expressed as abstract state machines, ASMs, where errors were found) and has also developed a collection of PVS theories for reasoning about operational and denotational semantics in this context. Another application related to programming language implementation is the security of Java style dynamic linking [10].

11 Summary

The applications sketched above give an idea of the range of projects for which PVS has been used and also provide a resource for those undertaking similar work. Additional descriptions can be found in the PVS Bibliography, which provides over 250 citations [38].

The development of PVS has been strongly influenced by practical applications and by feedback from users, and we expect this to continue. Enhancements currently in progress include direct and very fast execution for a substantial subset of the PVS language (this supports computational reflection [46], as well as improved validation of specifications [16]), and faster and more automated theorem proving. Those planned for the near future include support for refinement and a more open system architecture.

References

1. Parosh Aziz Abdulla, Aurore Annichini, Saddek Bensalem, Ahmed Bouajjani, Peter Habermehl, and Yassine Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In Halbwachs and Peled [17], pages 146–159.
2. Andren Alborghetti, Angelo Gargantini, and Angelo Morzenti. Providing automated support to deductive analysis of time critical systems. In Mehdi Jazayeri and Helmut Schauer, editors, *Software Engineering—ESEC/FSE '97: Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1301 of *Lecture Notes in Computer Science*, pages 211–226, Zurich, Switzerland, September 1997. Springer-Verlag.
3. Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.
4. Myla Archer and Constance Heitmeyer. Mechanical verification of timed automata: A case study. In *IEEE Real-Time Technology and Applications Symposium (RTAS'96)*, pages 192–203, Brookline, MA, June 1996. IEEE Computer Society.

5. Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [3], pages 323–335.
6. Bettina Buth. PAMELA + PVS. In Michael Johnson, editor, *Algebraic Methodology and Software Technology, AMAST'97*, volume 1349 of *Lecture Notes in Computer Science*, pages 560–562, Sydney, Australia, December 1997. Springer-Verlag.
7. Ricky W. Butler and Jon A. Sjogren. A PVS graph theory library. NASA Technical Memorandum 1998-206923, NASA Langley Research Center, Hampton, VA, February 1998.
8. Raphaël Couturier and Dominique Méry. An experiment in parallelizing an application using formal methods. In Hu and Vardi [24], pages 345–356.
9. Judith Crow and Ben L. Di Vito. Formalizing Space Shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, July 1998.
10. Drew Dean. Static typing with dynamic linking. In *Fourth ACM Conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, April 1997. Association for Computing Machinery.
11. Darryl Dieckman, Perry Alexander, and Philip A. Wilsey. ActiveSPEC: A framework for the specification and verification of active network services and security policies. In Nevin Heintze and Jeannette Wing, editors, *Workshop on Formal Methods and Security Protocols*, Indianapolis, IN, June 1998. Informal proceedings available at `http://www.cs.bell-labs.com/who/nch/fmsp/program.html`.
12. Bruno Dutertre and Victoria Stavridou. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering*, 23(5):267–278, May 1997.
13. Steve Easterbrook, Robyn Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1):4–14, January 1998.
14. Simon Fowler and Andy Wellings. Formal development of a real-time kernel. In *Real Time Systems Symposium*, pages 220–229, San Francisco, CA, December 1997. IEEE Computer Society.
15. Ganesh Gopalakrishnan and Phillip Windley, editors. *Formal Methods in Computer-Aided Design (FMCAD '98)*, volume 1522 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1998. Springer-Verlag.
16. David Greve. Symbolic simulation of the JEM1 microprocessor. In Gopalakrishnan and Windley [15], pages 321–333.
17. Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.
18. David Hardin, Matthew Wilding, and David Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Hu and Vardi [24], pages 39–44.
19. Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.
20. Mats P. E. Heimdahl and Barbara J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *IEEE High-Assurance Systems Engineering Workshop (HASE '96)*, pages 252–262, Niagara on the Lake, Canada, October 1996.
21. John Hoffman and Charlie Payne. A formal experience at Secure Computing Corporation. In Hu and Vardi [24], pages 49–56.

22. Jozef Hooman. Compositional verification of real-time applications. In Willem-Paul de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference (Revised lectures from International Symposium COMPOS'97)*, volume 1536 of *Lecture Notes in Computer Science*, pages 276–300, Bad Malente, Germany, September 1997. Springer-Verlag.
23. Ravi Hosabettu, Mandayam Srivas, and Ganesh Gopalakrishnan. Proof of correctness of a processor with reorder buffer using the completion functions approach. In Halbwachs and Peled [17], pages 47–59.
24. Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.
25. Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrick Tews. Reasoning about Java classes. In *Proceedings, Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 329–340, Vancouver, Canada, October 1998. Association for Computing Machinery. Proceedings issued as ACM SIGPLAN Notices Vol. 33, No. 10, October 1998.
26. Pertti Kellomäki. Verification of reactive systems using DisCo and PVS. In *Formal Methods Europe FME '97*, volume 1313 of *Lecture Notes in Computer Science*, pages 589–604, Graz, Austria, September 1997. Springer-Verlag.
27. Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 64–78, Palo Alto, CA, November 1996. Springer-Verlag.
28. Abdel Mokkedem, Ravi Hosabettu, and Ganesh Gopalakrishnan. Formalization and proof of a solution to the PCI 2.1 bus transaction ordering problem. In Gopalakrishnan and Windley [15], pages 237–254.
29. César Muñoz. PBS: Support for the B-method in PVS. Technical Report SRI-CSL-99-1, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1999.
30. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur and Henzinger [3], pages 411–414.
31. Sam Owre, John Rushby, and N. Shankar. Integration in PVS: Tables, types, and model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 366–383, Enschede, The Netherlands, April 1997. Springer-Verlag.
32. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
33. Seungjoon Park and David L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.
34. N.S. Pendharkar and K. Gopinath. Formal verification of an O.S. submodule. In V. Arvind and R. Ramanujin, editors, *18th Conference on the Foundations of Software Technology and Theoretical Computer Science*, volume 1530 of *Lecture Notes in Computer Science*, pages 197–208, Madras, India, December 1998. Springer-Verlag.
35. Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal verification for time-triggered clock synchronization. In Weinstock and Rushby [48], pages 207–226.

36. Amir Pnueli and Tamara Arons. Verification of data-insensitive circuits: An in-order-retirement case study. In Gopalakrishnan and Windley [15], pages 351–368.
37. S. P. Rajan, M. Fujita, K. Yuan, and M. T-C. Lee. ATM switch design by high level modeling, formal verification, and high level synthesis. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 3(4):554–562, October 1998.
38. John Rushby. PVS bibliography. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA. Constantly updated; available at `http://www.csl.sri.com/pvs-bib.html`.
39. John Rushby. Formal methods and their role in the certification of critical systems. In Roger Shaw, editor, *Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop)*, pages 1–42, Bruges, Belgium, September 1995. Springer. Also to be issued as part of the *FAA Digital Systems Validation Handbook* (the guide for aircraft certification).
40. John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.
41. Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [17], pages 443–454.
42. Jens U. Skakkebæk and N. Shankar. Towards a Duration Calculus proof assistant in PVS. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 660–679, Lübeck, Germany, Sept. 1994. Springer-Verlag.
43. David W. J. Stringer-Calvert. *Mechanical Verification of Compiler Correctness*. PhD thesis, University of York, Department of Computer Science, York, England, March 1998. Available at `http://www.csl.sri.com/~dave_sc/papers/thesis.html`.
44. Kothanda Umamageswaran, Krishnan Subramani, Philip A. Wilsey, and Perry Alexander. Formal verification and empirical analysis of rollback relaxation. *Journal of Systems Architecture (formerly published as Microprocessing and Microprogramming: the Euromicro Journal)*, 44(6–7):473–495, March 1998.
45. Ben L. Di Vito. A model of cooperative noninterference for integrated modular avionics. In Weinstock and Rushby [48], pages 269–286.
46. Friedrich von Henke, Stephan Pfab, Holger Pfeifer, and Harald Rueß. Case studies in meta-level theorem proving. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*, pages 461–478, Canberra, Australia, September 1998. Springer-Verlag.
47. M. Wahab. Verification and abstraction of flow-graph programs with pointers and computed jumps. Research Report CS-RR-354, Department of Computer Science, University of Warwick, Coventry, UK, November 1998. Available at `http://www.dcs.warwick.ac.uk/pub/reports/rr/354.html`.
48. Charles B. Weinstock and John Rushby, editors. *Dependable Computing for Critical Applications—7*, volume 12 of *Dependable Computing and Fault Tolerant Systems*, San Jose, CA, January 1999. IEEE Computer Society.
49. Matthew M. Wilding, David S. Hardin, and David A. Greve. Invariant performance: A statement of task isolation useful for embedded application integration. In Weinstock and Rushby [48], pages 287–300.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Mechanizing Formal Methods: Opportunities and Challenges*

Reprint of an invited paper presented at the 9th International Conference of Z Users (ZUM '95, Springer Verlag LNCS 967, pp. 105–113), Limerick, Ireland, September 1995

John Rushby

Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA

Abstract

Mechanization makes it feasible to calculate properties of formally specified systems. This ability creates new opportunities for using formal methods as an exploratory tool in system design. Achieving enough efficiency to make this practical raises challenging problems in automated deduction. These challenges can be met only by approaches that integrate consideration of its mechanization into the design of a specification language.

1 Introduction

All formal methods rest on the conviction that requirements and designs for computer systems and software can be modeled mathematically, and that many questions concerning properties of those requirements and designs can then be settled by calculation. Advocates of formal methods differ, however, in the extent to which they stress the conceptual and methodological aspects of these methods, as opposed to their calculational aspects.

While appreciating the methodological benefits of mathematical concepts and notations, I believe that the distinctive merit of specifically *formal* methods is that they support calculation. Using automated deduction (i.e., theorem proving) and related techniques (e.g., model checking) it is possible to calculate whether a formally-described design satisfies its specification or possesses certain properties. To be useful, it must be possible to perform these calculations for specifications and properties of practical interest with reasonable ease and efficiency.

It takes a lot of theorem-proving power to do this, and mechanized specification languages must be designed to mesh well with the most effective theorem-proving techniques. For example, reasoning about equality in the presence of uninterpreted

*This work was partially supported by the Air Force Office of Scientific research under contract F49620-95-C0044.

function symbols is crucial to most applications of mechanized formal methods, and efficient techniques for achieving this (such as congruence closure [14]) require that functions are total. However, it is draconian and rather unnatural to force all functions (including, for example, division) to be total, so an effectively mechanized formal method requires careful and integrated design choices to be made for both the specification language and its supporting mechanization. In this particular case, it is necessary to find some reasonably attractive treatment for partial functions that does not compromise the efficiency of equality reasoning.

While developing such a treatment poses a challenge, the hypothesized availability of a powerful theorem prover creates new opportunities: for example, the typechecker of our specification language could use the theorem prover to check that partial functions are never applied outside their domains. This balancing of challenges and opportunities, and the corresponding need for integrated design decisions, arises again and again when mechanizing formal methods. In the following sections, I will briefly describe the main opportunities created by mechanized formal methods, and the technical challenges in achieving effective mechanization. My perspective on these topics is influenced by experiences in the development and use of PVS [9].

2 Opportunities for Making Effective Use of Mechanized Formal Methods

It is often assumed that the main goal of mechanized formal methods is "proving correctness" of programs or detailed hardware designs, and this assumption may be reinforced by the term "verification system" that is commonly used to describe mechanized tools for formal methods. In fact, however, this assumption is wrong on both counts: most advocates of mechanized formal methods consider such early-lifecycle products as requirements, architectural designs, and algorithms to be more attractive targets for their tools than finished programs or gate layouts, and their focus is at least as much on finding faults and on design exploration as it is on verifying correctness.

Preference for early-lifecycle applications of mechanized formal methods is partly a consequence of the strength of traditional methods for late-lifecycle activities. Traditional methods for the development and quality assurance of program code and detailed hardware designs are sufficiently effective that very few significant faults are introduced and remain undetected at these late stages of the lifecycle (for example, of 197 critical faults found during integration testing of two JPL spacecraft, only three were programming mistakes [8]), and only a small fraction of overall development costs (typically, less than 10%) are incurred here. In contrast, mechanized formal methods are quite expensive to apply at these stages. Primarily, this is because the products of the late lifecycle are usually large—typically, hundreds of thousands, or even millions, of gates or lines of code—and their sheer size makes formal verification costly.

But if traditional methods are effective in the later stages of the lifecycle, the same cannot be said of the earlier stages. Natural-language, diagrams, pseudocode, and other conventional ways for describing requirements and preliminary designs do not support calculation, so the main way to deduce their properties and consequences is through the fallible processes of inspection and review. The fallibility of these processes is illustrated by the JPL data cited earlier: since only three of the 197 critical faults were due to programming errors, it follows that the other 194 were introduced at earlier stages. Lutz reports that 50% of these faults were due to flawed requirements (mainly omissions) for individual components, 25% were due to flawed designs for these components, and the remaining 25% were due to flawed interfaces between components and incorrect interactions among them [8].

Rapid prototyping and simulation provide more repeatable and systematic examination of these issues, but often force premature consideration of implementation questions and thereby divert attention from the most important topics. Mechanized formal methods, on the other hand, can support direct analysis and exploration of the products of the early lifecycle: as soon as we have written down a few logical formulas that describe some aspect of the system, so we can begin to check their consequences—such as whether they entail some expected property, or are mutually contradictory.

The purposes for which mechanized formal methods may be used in the early lifecycle are as much those of validation and exploration as verification: the opportunities are to validate requirements specifications, to explore different system architectures and the interactions of their components, to debug critical algorithms and to understand their properties and assumptions, and to cope with changes.

Mechanized formal methods can assist requirements validation by checking whether a formal statement of requirements entails other expected properties: intuitions such as "if I've got that right, then this ought to follow," can be examined in a formal manner using theorem proving or model checking. While confirmation of an expected property is gratifying, a more common outcome—at least in the early stages—is the discovery that the requirements must be revised, or that some new assumption must be adopted. The rigor of mechanized analysis renders the discovery of such oversights far more systematic than is the case for informal reviews. Furthermore, mechanization allows us to check, rapidly and reliably, that previously examined properties remain true following each revision to the requirements.

The assurance derived by checking that expected properties are entailed by a requirements specification may be specious if the requirements are inconsistent (i.e., mutually contradictory): every property is entailed by an inconsistent specification. Consistency of formal specifications can be demonstrated by exhibiting a model; an equivalent demonstration can be checked mechanically using theory interpretations: the basic idea is to establish a translation from the types and constants of the "source" specification (the one to be shown consistent) to those of a "target" specification and to show that the axioms of the source specification, when translated into the terms of the target specification, become provable theorems of that target specification. If this can be done, then we have demonstrated *relative* consistency:

the source specification is consistent if the target specification is. Generally, the target specification is one that is specified definitionally, or one for which we have some other good reason to believe in its consistency.

Following requirements validation, mechanized formal methods can be used to explore candidate system architectures. Architectures consist of interacting components; the concerns at this stage are generally to verify that the properties assumed of the components and of their interaction are sufficient to ensure satisfaction of the overall requirements. The chief benefits of applying mechanically checked formal verification to this task are the ability to explore alternative designs and assumptions, and to prune unnecessary assumptions. For example, formal examination of an architecture for fault masking and transient recovery in flight control systems reveals the need for interactive consistency on sensor inputs [11]. This can be achieved by a Byzantine Agreement algorithm [5]. Inputs to the majority vote function must also satisfy interactive consistency and it may therefore appear as if these, too, need to be run through a Byzantine Agreement algorithm. In fact, this is not required: it is possible to prove that interactive consistency of voter inputs is an inherent property of the architecture. Mechanized formal analysis allows this attribute of the architecture to be determined with certainty, and it also allows determination of the exact circumstances under which a modified architecture provides recovery from transient faults [11].

As with requirements validation, exploration of architectural design choices is an iterative process that is greatly facilitated by the rigor and repeatability of mechanized formal methods. Simulation and direct execution share the repeatability of formal methods but can examine only a few cases, whereas deductive formal methods allow consideration of *all* cases. But, unfortunately, the price of this generality can be less automated and more difficult analysis: we may need to invent invariants and to undertake proofs by induction in order to cover an infinite state space. Often, however, an effective alternative is available: we may be able to abstract or "downscale" a specification to a finite state space that can be examined, exhaustively and automatically, by model checking or by explicit state exploration. The great automation of formal finite-state methods creates new opportunities for applying formal methods in the design loop. Experience indicates that examining *all* the cases of such an abstracted system description is generally more effective at finding faults than testing or simulating *some* of the cases of the full description [3].

Once a preferred system architecture has been selected we may, recursively, explore architectures for its components or—once a sufficiently detailed level has been reached—investigate algorithms for those components. As with the earlier stages, the great benefits of using mechanized formal methods to examine algorithms are the abilities to explore alternatives, to prune assumptions, and to adapt to design changes. For example, the journal presentation of the interactive convergence clock synchronization algorithm [4] has an assumption that all initial clock adjustments are zero. Friedrich von Henke and I retained this assumption when we formally verified the algorithm [13]. Subsequently, when contemplating design of a circuit to implement part of the algorithm, it became clear that this assumption is exceedingly

inconvenient. I explored the conjecture that it is unnecessary by simply striking it out of our formal specification and rerunning the proofs of all the lemmas and theorems that constitute the verification. (There are about 200 proofs in the full verification and it takes about 10 minutes for the theorem prover to check them all.) It turned out that the proofs of a few lemmas failed without the assumption, but examination showed that those lemmas could easily be restated, or given different proofs. A few hours of work were sufficient to make these adjustments to the formal specification and mechanically checked verification.

In other revisions to this algorithm and its verification, I have tightened the bound on the achieved clock skew, and extended the fault model so that the algorithm tolerates larger numbers of simple faults, without compromising its ability to resist arbitrary (i.e., Byzantine) faults [12]. In each case, the effort required to investigate the proposed revision and to rework the formal specification and verification was on the order of a day or two. In another example, Lincoln and I developed the formal specification and verification of a Byzantine Agreement algorithm for an asymmetric architecture in less than a day by modifying an existing treatment for a symmetric architecture [7].

The ability to make these enhancements to complex algorithms, rapidly and reliably, is an opportunity created by mechanized formal methods. Informal methods of proof are unreliable in these domains (see [6, 9, 13] for examples) and it requires superhuman discipline to bring the same level of care and skepticism to the scrutiny of a modified algorithm as to the original. A formal specification and verification, on the other hand, is a reusable intellectual resource: its properties can be calculated, and those calculations can be mechanized.

In summary, the opportunities created by mechanized formal methods are similar to those offered by mechanized calculation in other fields, such as computational fluid dynamics: exploration of design alternatives, early detection of design errors, identification of assumptions, the ability to analyze the consequences of changes and responses to them, and the acquisition of an enhanced understanding that can lead to further improvements. In the next section, I will briefly examine some of the challenges in developing mechanizations of formal methods that can make these opportunities reality.

3 Technical Challenges in Mechanizing Formal Methods

The primary challenge to achieving the benefits described above is the difficulty of mechanizing formal deduction in a way that is both efficient and enlightening. Since automated deduction—i.e., theorem proving—is a fairly well-developed field, applying this technology to formal methods might seem to be simply a matter of engineering. In fact, most of the challenges are those of engineering, but they are not simple. Theorem proving in support of formal methods raises issues that are quite different from those that have traditionally been of interest in the theorem-proving

community. Most notably, candidate theorems generated by formal methods in their exploratory and debugging roles are very likely to be false. In these cases, mechanization is expected to quickly reveal the falsehood, and to help identify its causes. The traditional concern of theorem proving, however, has been to prove true theorems, and most off-the-shelf theorem provers are therefore ill-suited to the needs of formal methods.

But although an existing theorem prover is unlikely to be useful in support of formal methods, we must not ignore the component techniques developed for automated theorem proving. One of the principal reasons that many attempts at mechanizing formal methods have failed is that their developers did not appreciate the raw power and speed that is needed from their theorem-proving components, and did not make use of the relevant techniques. The most important of these techniques are decision procedures for specialized, but ubiquitous, theories such as arithmetic, equality, function updates (i.e., overriding), and propositional calculus. Decision procedures are helpful in discovering false theorems (especially if they can be extended to provide counterexamples) as well as in proving true ones, and their automation dramatically improves the efficiency of proof.

There are decision procedures for many useful theories, but few problems fall precisely in the domain of any single one of them, so one of the big engineering challenges in mechanizing formal methods is to develop effective combinations of decision procedures. This requires very careful selection of the individual procedures. For example, the decision procedure for Presburger arithmetic (i.e., the first order theory of linear arithmetic with relation symbols such as $<$) does not consider uninterpreted function symbols. Since function symbols are pervasive in formal specifications, a better choice than true Presburger arithmetic is the theory of ground (i.e., unquantified) linear arithmetic, which can be combined with the theory of equality over uninterpreted function symbols [15].

Small extensions to decidable theories can have considerable value. For example, it is not possible to add full nonlinear multiplication to the decision procedures for linear arithmetic, but it is possible to add the ability to reason about the signs of products (e.g., "a minus times a minus is a plus") and this proves to be a significant benefit in practice.

Rewriting is another technique that is essential to efficient mechanization of formal methods. Unrestricted rewriting provides a decision procedure for theories axiomatized by terminating and confluent sets of rewrite rules, but few such theories arise in practice. Consequently, rewriting cannot be unrestricted, in general, but must be performed under some control strategy. For example, one of the control strategies used in PVS will rewrite a definition whose body involves a top-level *if-then-else* only if the condition to the *if* can be reduced to *true* or *false*. This reduction may involve use of decision procedures and further rewriting, and it is possible to expend considerable resources on a search that is ultimately unsuccessful (because it does not succeed in reducing the condition). These resources will not have been wasted, however, if they allow the theorem prover to avoid making an unprofitable rewrite: in most contexts, the heuristic effectiveness of a good control

strategy is likely to be more beneficial than the raw speed of a blind rewriter. As this description suggests, rewriting and decision procedures cannot stand apart: truly effective theorem provers must integrate them very tightly. A classic account of the issues in such integration is given by Boyer and Moore [1].

Integration is a pervasive theme in the effective mechanization of formal methods: many individual techniques work well on selected examples, but fail in more realistic contexts because problems seldom fall exactly within the scope of one method. Sometimes the integration must be tight, as in cooperating decision procedures, and the integration of decision procedures with rewriting. In other cases, the integration can be less tight; a good example is model checking within a theorem-proving context. Whereas theorem proving attempts to show that a formula follows from given premises, model checking attempts to show that a given system description is a model for the formula. As noted in the previous section, an advantage of model checking is that, for certain finite state systems and temporal logic formulas, it is much more automatic and efficient than theorem proving. The additional benefits of a system that provides both theorem proving and model checking are that model checking can be used to discharge some cases of a larger proof or, dually, that theorem proving can be used to justify the reduction to finite state that is required for automated model checking [10].

Model checking can provide a further benefit: before undertaking a potentially difficult and costly proof, we may be able to use model checking to examine some restricted or special cases. Any errors that can be discovered and eliminated in this way will save time and effort in theorem proving. This is a particular case of a more general desideratum: mechanized formal methods should provide a graduated collection of tools and techniques that apply increasingly strict scrutiny at correspondingly increasing cost. Representative techniques include typechecking, animation or direct execution, model checking, and theorem proving. These techniques become more effective if they are integrated so that each can use capabilities of the others. For example, typechecking can be made more strict if it is allowed to use theorem proving, rather than being restricted to trivially decidable properties; certain specifications can be executed on test cases by using the theorem prover to perform rewriting (in this case, fast "blind" rewriting may be desirable); and theorem proving can be more efficient if it makes use of type information provided by the typechecker.

Achieving the necessary integrations involves more than careful engineering of theorem proving and support tools: it extends to the design of the specification language itself. As noted in the introduction, efficient equality reasoning, for example, requires that functions are total. If we allow the concerns of mechanization to dominate language design, we may then decide that our specification language should provide only total functions. Similar considerations may lead us to restrict quantification to first-order (or to eliminate explicit quantification altogether), to restrict recursive definitions to the syntactic form of primitive recursion, or to require all formulas to be equations. In the limit, we may provide a raw logic devoid of the features expected of a specification language. Conversely, concerns for an

expressive notation may lead us to provide a specification language that cannot be mechanized effectively. This is not to say that expressiveness cannot be combined with mechanization, but that expressiveness must not be considered in isolation from mechanization. For this reason, I consider it dangerous to look to the classical foundations of mathematics for guidance when designing a specification language. These formal systems (notably, first-order logic with axiomatic set theory) were created in order to be studied, not in order to be used—the"...interest in formalized languages being less often in their actual and practical use as languages than in the general theory of such use and its possibilities in principle" [2, page 47]. Unsurprisingly, therefore, set theory has characteristics that pose difficulty for mechanization—for example, as already noted, functions are inherently partial in set theory (they are sets of pairs). Also, it is difficult to provide really strict typechecking (and hence, early error detection) for set theory without sacrificing some of its flexibility: for example, a function is a set, so it can (sometimes) make sense to form its union with another set, and it is therefore not clearcut whether type restrictions should prohibit or allow this sort of construction.

If mechanization is a goal, then design of the specification language and selection of its underlying logic should not be undertaken independently of consideration of its mechanization. This does not mean that concern for mechanization must inevitibly restrict or impoverish a notation—rather, I believe that availability of powerful mechanization creates new opportunities for the design of expressive, yet mechanically tractable, notations. An example is provided by the treatment of partial functions such as division in PVS, whose specification language supports the notion of *predicate subtypes*. These allow the nonzero real numbers to be defined as follows.

```
nonzero_real: TYPE = { r: real | r /= 0 }
```

We can then give the signature of division as

```
/: [real, nonzero_real → real]
```

and the function is total on this precisely specified domain. We can then state and prove the following result.

```
inverse_sum: LEMMA
   ∀ (a, b: real):
     a /= 0 ∧ b /= 0 ⊃ (1/a + 1/b) = (a + b)/(a × b)
```

PVS allows a value of a supertype to appear where one of a subtype is required, provided the value can be proved, in its context, to satisfy the defining predicate of the subtype concerned. In this case, PVS will generate the following three proof obligations, called Type Correctness Conditions (TCCs), that must be discharged before `inverse_sum` is considered fully type-correct. The three TCCs correspond to the three appearances of division in the formula `inverse_sum` and collectively ensure that the value of the formula does not require division by zero. The first two TCCs

can be discharged automatically by a decision procedure for linear arithmetic; the third requires extensions mentioned earlier that reason about the signs of nonlinear products.

tcc1: OBLIGATION $\forall$ (a, b: real): $a \neq 0 \wedge b \neq 0 \supset a \neq 0$

tcc2: OBLIGATION $\forall$ (a, b: real): $a \neq 0 \wedge b \neq 0 \supset b \neq 0$

tcc3: OBLIGATION $\forall$ (a, b: real): $a \neq 0 \wedge b \neq 0 \supset (a \times b) \neq 0$

Predicate subtypes in PVS provide many more capabilities than are suggested by this simple example: in particular, injections and surjections are defined as predicate subtypes of the functions, and state-machine invariants can be enforced by the same mechanism. The source of these conveniences and benefits is allowing typechecking to require theorem proving (i.e., to become algorithmcally undecidable), which is only feasible if a powerful and automated theorem prover is assumed to be available. Conversely, the power of the theorem prover is enhanced by the precision of the type information that is provided by the language and its typechecker.

4 Conclusion

Mechanization creates new opportunities for formal methods: by making it feasible to calculate properties of formally specified designs, mechanization allows exploration of alternative designs, examination of assumptions, adaptation to changed requirements, and verification of desired properties. These opportunities are likely to have maximum benefit when applied early in the development lifecycle, and to the hardest and most important problems of design.

To realize these benefits, mechanizations of formal methods must provide several capabilities ranging from very strict typechecking, to powerfully automated theorem proving. These individual capabilities need to be closely integrated with each other, and with the specification language. Because of the integration required, consideration of its mechanization must be factored into the design of a specification language.

References

[1] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. In *Machine Intelligence*, volume 11. Oxford University Press, 1986.

[2] Alonzo Church. *Introduction to Mathematical Logic*, volume 1 of *Princeton Mathematical Series*. Princeton University Press, Princeton, NJ, 1956. Volume 2 never appeared.

[3] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992. Cambridge, MA, October 11-14.

[4] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[5] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[6] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

[7] Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model. In *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, pages 107–120, Gaithersburg, MD, June 1994. IEEE Washington Section.

[8] Robyn R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, January 1993.

[9] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[10] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[11] John Rushby. A fault-masking and transient-recovery model for digital flight-control systems. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, chapter 5, pages 109–136. Kluwer, Boston, Dordecht, London, 1993. An earlier version appeared in [16, pp. 237–257].

[12] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Los Angeles, CA, August 1994. Association for Computing Machinery.

[13] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[14] Robert E. Shostak. An algorithm for reasoning about equality. *Communications of the ACM*, 21(7):583–585, July 1978.

[15] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.

[16] J. Vytopil, editor. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, January 1992. Springer-Verlag.

Invited paper presented to joint session of CAV and CADE '96. Appears in proceedings of CAV '96, Springer Verlag LNCS 1102, pp. 169–183, July 1996

Automated Deduction and Formal Methods*

John Rushby

Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA

Abstract. The automated deduction and model checking communities have developed techniques that are impressively effective when applied to suitable problems. However, these problems seldom coincide exactly with those that arise in formal methods. Using small but realistic examples for illustration, I will argue that effective deductive support for formal methods requires cooperation among different techniques and an integrated approach to language, deduction, and supporting capabilities such as simulation and the construction of invariants and abstractions. Successful application of automated deduction to formal methods will enrich both fields, providing new opportunities for research and use of automated deduction, and making formal methods a truly useful and practical tool.

1 Introduction

Formal methods are a natural application area for automated deduction—yet, with few exceptions, tools for mainstream formal methods provide little more than rudimentary support for deduction, and few theorem provers find application in formal methods. Model checking and related techniques are gaining acceptance in important specialized areas, but have yet to penetrate the larger field. This disconnect between formal methods and the very technologies that could help increase its utility and appeal is unfortunate, and deserves explanation and remedy.

My opinion is that many techniques for automated deduction (and for simplicity I include model checking under this heading) provide excellent solutions to individual problems, but that formal methods require more integrated approaches to provide solutions that are effective across a broad range of problems. In the following sections, I outline some prototypical applications of formal methods and suggest some of the capabilities required of automated deduction if it is to achieve more widespread use in this area. I discuss these topics under three headings: language, theories, and interaction in the sections that follow. Brief conclusions are presented in Section 5.

* This work was supported by the Air Force Office of Scientific Research under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931. The applications described were undertaken for NASA Langley Research Center under contracts NAS1-18969 and NAS1-20334 and for ARPA through NASA Ames Research Center under contract NASA-NAG-2-891.

2 Language

By *formal methods* I mean the use of techniques derived from mathematical logic for the specification and analysis of computational systems. There are two elements here: *specification*, by which I mean a descriptive activity in which logical notation is valued for its contributions to both the intellectual process of design and the communication of designs, and *analysis*, by which I mean systematic and repeatable methods for deducing properties of specifications and of the designs that they represent. Automated deduction has obvious relevance in the mechanization of analysis, but formal methods practitioners attach great importance to specification and are unwilling to compromise on the convenience of expression provided by a full specification language. To achieve acceptance, it therefore seems necessary that automated deduction should be harnessed to rather rich notations.

To suggest some of the capabilities desired, I outline a typical "requirements specification" for a function in the Space Shuttle's control system called "Jet-Select" [6]. This function is responsible for selecting which of the Shuttle's Reaction Control System (RCS) jets (or thrusters) should be fired in order to accomplish a given translational or rotational acceleration. I will concentrate on the "Vernier/Alt" component for rotation, which can operate in one of two modes: in Vernier mode, only the small "vernier" jets are considered for selection; in Alt (alternative) mode, only the larger "primary" jets are considered. The basic Jet-Select calculations are the same whether in Vernier or Alt mode, except that the six vernier jets are treated singly, while the 38 primary jets are treated in groups. (The primary jets are arranged in 14 groups, each consisting of two, three, or four jets located adjacent to each other and firing in the same direction; only 11 of the 14 groups are useful for rotational maneuvers.) In Vernier mode, Jet-Select chooses up to three individual vernier jets to fire, whereas in Alt mode it selects up to three groups of primary jets, and then selects exactly one jet from each of the chosen groups. (The jets within each group are ranked in a priority order and it is the available jet of highest priority that is fired when its group is selected in Alt mode.) Various vernier jets and groups of primary jets are excluded from consideration in certain submodes (e.g., jets whose plumes extend into the area above the cargo bay are excluded in "low +z" mode) and individual jets may be marked "unavailable" due to failure or by crew selection.

The selection of vernier jets or primary groups is performed by an algorithm known as "max dot-product" (this particular exercise in formalization was undertaken in preparation for introduction of a new algorithm called "min angle"). For each vernier jet and primary group, a table records the rotational velocity vector imparted by firing that jet (or a member of that group) for a standard period. (Actually, there are several tables, parameterized by whether there is a payload attached to the Shuttle's robotic arm, and where the arm is positioned.) The algorithm proceeds by first selecting the vernier jet or primary group whose acceleration has the largest scalar (dot) product with the rotational acceleration vector actually desired; the second and third jets (if required) are similarly selected as those with the second and third largest scalar products, *provided* the

dot-product of the second exceeds some fraction t_1 of the first, and that of the third exceeds some fraction t_2 of the second.

The major goal here is to use formal methods to specify the desired functionality as clearly as possible. The role of automated deduction in this example is to contribute to validation of the specification by examining putative "challenge" theorems such as "a failed jet will never be selected."

A good specification for this component of Jet-Select should make clear that the max dot-product algorithm is essentially the same in both Vernier and Alt modes, except that in the former it operates over individual vernier jets, while in the latter it operates over groups of primary jets. This argues for a specification notation that provides parameterized theories so that specification of the same algorithm can be instantiated over these different domains. Although not exemplified by Jet-Select, many applications of formal methods also require parameterized representations for standard computer science data structures such as lists, trees, and arrays.

Next, we can observe that the output of Jet-Select is most naturally considered as a set of jets, and the groups of primary jets are also naturally considered as sets. Thus, our specification notation should incorporate a representation for sets. Most practitioners of formal methods prefer their specification notation to be strongly typed, and this particular application seems to call for subtyping: surely the vernier and primary jets are naturally considered as subtypes of the type of all jets. But then the output of the algorithm will be either a set of vernier jets or a set of primary groups (the latter is then converted to a set of primary jets), whereas the output of Jet-Select as a whole must be a set of jets. Hence, our specification notation must somehow extend the subtyping relation between (for example) vernier jets and all jets to a compatible subtyping relation between the *sets* of such jets.

There are (at least) two ways to specify that the (intermediate) result of Jet-Select should be the set of vernier jets or primary groups satisfying the max dot-product criterion. One way would simply axiomatize the desired property, the other would attempt to represent the algorithm suggested by the informal description (i.e., the iterative selection of the three best jets or groups from among those available). The latter approach might require the specification language to incorporate a treatment of imperative programs. It would also require a way to identify the jet or group in a given set that has the maximum dot-product. For generality, we might like to provide a library axiom defining the maximum of a set to be its largest member with respect to some given ordering. This is most directly accomplished by quantification, but we must ensure that the ordering relation has the appropriate algebraic properties and must take proper care of the case where the set is empty, or risk unsoundness. A specification language should help ensure that these obligations are not overlooked.[2]

Most theorem provers support raw logics that lack the notational conveniences mentioned above. In my experience, it quite hopeless to persuade users

[2] For example, the PVS declaration

$$max(s : \text{setof}[T]) : \{t : T | t \in s \wedge \forall (x : T) : x \in s \supset (t > x \vee x = t)\}$$

of formal methods (let alone those who are not yet users) to adopt such impoverished notations. To observe that it is perfectly *feasible* to provide a specification for Jet-Select in quite primitive logics (e.g., those without quantification) misses the point—this simply is not what users of formal methods want to do.

Left to their own devices, users of formal methods develop or adopt notations such as B, VDM, RAISE, or Z. These make few concessions to the needs of efficient automated deduction and the tools that have been developed for them provide little more than interactive proof checking unsupported by significant automation (e.g., [8, 10]). I have argued elsewhere [14] that choices made in the designs of these languages (e.g., in the case of Z, set theory with partial functions, and no notion of definition) are inimical to automated deduction, and that really efficient deductive support is therefore unlikely to be forthcoming for them.

One of the challenges to those who would provide automated deduction for formal methods is therefore to contribute to the design of specification languages that combine the felicity of expression desired for formal methods with the possibility of powerfully automated support. Rather than being a limitation on specification language design, I believe that closer integration of language and automated deduction can have a liberating effect—because it makes it possible to contemplate design choices that require theorem proving in typechecking. We have exploited this opportunity to some extent in PVS [12] (where subtyping, for example, can generate proof obligations) but many further opportunities remain.

It is not necessary that the logic supported by a theorem prover should *be* a full specification language, but there must be some translation from the latter to the former. Furthermore, the translation must be maintained during interaction with the prover: it is unlikely to be acceptable if proof of a conjecture expressed in the specification language must be conducted in terms of its translation into the primitives of the underlying logic.

3 Theories

Automated deduction must not only support the rich linguistic capabilities desired in formal methods, but must also provide very effective automation for theories that are commonly encountered.

For illustration, I will use a verification of the Interactive Convergence Algorithm for Byzantine fault-tolerant clock synchronization [9] that Friedrich von Henke and I performed some years ago [15]. The goal is to keep the clocks of distributed processors approximately synchronized, given that good clocks have some bounded drift rate, good processors can read the clocks of other good processors with some small error, and faulty processors and clocks are unconstrained (in particular, they can present conflicting information to different good processors). The clock of processor p is represented by an uninterpreted function $c_p(T)$

---

generates a proof obligation (to show that the type assigned to the value of *max* is inhabited) that can be discharged only if the set s is nonempty and $>$ is a well-ordering.

from "clock time" to "real time" (both interpreted as real numbers).[3] Clocks are adjusted every R clock time units (this duration is called a "frame" and the start time of the i'th frame is denoted $T^{(i)} = T^{(0)} + iR$), during a "synchronization period" of duration S clock time units occurring at the end of the frame. (The interval of the i'th frame is denoted $R^{(i)} = [T^{(i)}, T^{(i+1)}]$, and the i'th synchronization period is denoted $S^{(i)} = [T^{(i+1)} - S, T^{(i+1)}]$.) The adjustment to clock p for period i is $C_p^{(i)}$ clock time units and the adjusted clock for that period is denoted $c_p^{(i)}(T)$, where $c_p^{(i)}(T) = c_p(T + C_p^{(i)})$.

In the i'th synchronizing period, each processor p obtains an estimate $\Delta_{qp}^{(i)}$ of the skew between its clock and that of processor q. A parameter ϵ bounds the error in this estimate as follows.

Assumption A2. *If the clock synchronization conditions (defined below) hold for the i'th period, and processors p and q are nonfaulty through period i, then*

$$|\Delta_{qp}^{(i)}| \leq S$$

and

$$|c_p^{(i)}(T' + \Delta_{qp}^{(i)}) - c_q^{(i)}(T')| < \epsilon$$

for some time T' in $S^{(i)}$.

The algorithm is defined as follows.

Algorithm ICA. *For all processors p:*

$$C_p^{(i+1)} = C_p^{(i)} + \Delta_p^{(i)},$$

where

$$C_p^{(0)} \text{ is arbitrary,}$$

$$\Delta_p^{(i)} = \left(\frac{1}{n}\right) \sum_{r=1}^{n} \bar{\Delta}_{rp}^{(i)}, \quad \textbf{and}$$

$$\bar{\Delta}_{rp}^{(i)} = \textbf{if } |\Delta_{rp}^{(i)}| < \Delta \textbf{ then } \Delta_{rp}^{(i)} \textbf{ else } 0$$

and Δ is a clock time quantity that is a parameter to the algorithm.

The goal is to achieve the following *clock synchronization conditions*, provided that at most m processors (out of n) are faulty through period i, for real time constant δ and clock time constant Σ that are parameters to the algorithm.

Bounded skew: *If p and q are nonfaulty through period i, then*

$$|c_p^{(i)}(T) - c_q^{(i)}(T)| < \delta$$

for all T in $R^{(i)}$.

[3] A specification language with the ability to distinguish clock time and real time as different "dimensions" of the same type provides valuable additional error checking in these constructions.

Bounded adjustment: *If processor p is nonfaulty through period i, then*

$$|C_p^{(i+1)} - C_p^{(i)}| < \Sigma.$$

These conditions can be achieved, provided several assumptions (concerning, for example, the drift rate ρ of good clocks) are satisfied, together with several constraints on the parameters to the algorithm, such as the following.

Constraint C6. $\delta \geq 2(\epsilon + \rho S) + \frac{2m\Delta}{n-m} + \frac{n\rho R}{n-m} + \frac{n\rho\Sigma}{n-m} + \rho\Delta$

The proof depends on several lemmas, of which the following are among the most important.

Lemma 4. *If the clock synchronization conditions hold for i, processors p, q, and r are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]| < 2(\epsilon + \rho S) + \rho\Delta.$$

Lemma 5. *If the bounded skew clock synchronization condition holds for i, processors p and q are nonfaulty through period $i+1$, and $T \in S^{(i)}$, then*

$$|c_p^{(i)}(T) + \bar{\Delta}_{rp}^{(i)} - [c_q^{(i)}(T) + \bar{\Delta}_{rq}^{(i)}]| < \delta + 2\Delta.$$

The items of interest here are the theories involved: we have arithmetic expressions and relations involving both real and natural numbers, and both interpreted and uninterpreted function symbols. The ubiquity and complexity of the arithmetic used here are such that it would be intolerable to attempt verification of this algorithm without efficient deductive support for arithmetic. A library of lemmas and rewrite rules will not be adequate to the task: decision procedures are needed. The question then is: decision procedures for which theories? The importance of integer arithmetic is such that some tools for formal methods include decision procedures for Presburger arithmetic—that is the quantified theory of integer linear arithmetic. Since we have real numbers as well, a decision procedure for real closed fields might also seem appropriate. The problem with these choices is that we also have uninterpreted function symbols, which takes us outside these decidable theories. Inspection of various formulas appearing in the presentation of the algorithm shows that only Assumption A2 involves a nested quantifier (for T'), everything else is (implicitly) universally quantified at the outermost level. We can conclude that the quantifier reasoning here is likely to be easy, and we may therefore be prepared to deal with it outside the arithmetic decision procedures (either heuristically, or with user guidance). This will allow us to restrict the arithmetic decision procedures to just the ground case—where the combination of linear arithmetic with uninterpreted function symbols is decidable [4].

My experience with formal methods applications is that this tradeoff in favor of deciding ground theories is always worthwhile, since it allows the different decision procedures to be combined. Some theories, such as arithmetic, equality with

uninterpreted function symbols, and arrays[4] are so ubiquitous that decision procedures for their ground cases are essential for all productive work. Decision procedures for additional theories may be highly advantageous for particular classes of applications. For example, our experience with processor verification [17] has shown that the (large) library of rewrite rules used for the theory of bitvectors is the main impediment to effective automation, and we conjecture that a decision procedure for bitvectors would have a dramatic benefit. The development of new decision procedures for theories arising in formal methods is a valuable topic for research.

Important requirements for such decision procedures are the following.

- They must work cooperatively to decide the *combination* of their theories.
- They must deal gracefully with terms outside the decided theory. For example, the theory decided by the *SUP-INF* [16] and similar procedures is ground *linear* arithmetic, but several of the formulas used in clock synchronization contain nonlinear terms (and division). Although the full nonlinear case cannot be decided, it is important to deal with special properties (e.g., commutativity, and "a minus times a minus is a plus") without losing those properties that follow simply by treating nonlinear multiplication as uninterpreted. A similarly effective extension to division is also required. (Notice also that some treatment for the partiality of division by zero is needed; this may require coordination between the specification language and its deductive support—in PVS, for example, division by zero is excluded through type rules that generate proof obligations to show the divisor is nonzero.)
- Their behavior must be predictable. One of the strengths of decision procedures over heuristics is that the user should not have to puzzle over whether the failure to prove a conjecture is due to its falsehood, or an inadequate heuristic. This benefit is lost if the decided theory is not clearly characterized. And although performance is hard to guarantee given the super-exponential complexity of most decision procedures, "black holes" (where a small and apparently simple problem takes an inordinate amount of time) are to be avoided. Because they will form the inner loop of larger procedures, even linear speedups in the performance of decision procedures can have a dramatic impact on overall efficiency; more needs to be known about the relative practical performance of various decision procedures for the same problem, which anecdotal evidence indicates can differ by an order of magnitude or more [4]. Conjectures in formal methods applications often give rise to very large formulas, so it is crucial that decision procedures should be implemented in ways that scale reasonably well (using, for example, structure-sharing techniques similar to those in BDDs[5]).

[4] That is (in PVS notation) $f[(x) := y](z) = \textbf{if } z = x \textbf{ then } y \textbf{ else } f(z)$. This is also known as function updating or overriding.

[5] It goes without saying that propositional reasoning must be implemented very efficiently. Ordered binary decision diagrams (OBDDs) are the natural choice, but the Davis-Putnam procedure and the patented algorithm of Stålmarck [18] may be superior in some applications.

- Expressions that cannot be decided should be simplified. Especially in an interactive environment, it is important that the information presented to the user should be as brief and as simple as possible. But it should also be familiar—that is to say, expressions should retain, to the extent possible, the form they were originally given by the user, and should not be arbitrarily normalized. Simplification should merely eliminate redundancy, so that, for example, $(a+1)-1$, **if** *true* **then** a **else** b, and **if** B **then** a **else** a all become a; it should generally refrain from transformations such as that from $x \times (a+b)$ to $x \times a + x \times b$. One of the great advantages of decision procedures over heuristics is that they are sensitive only to the content and not to the form of expressions, so that syntactic representations can be chosen for the convenience of the user rather than the prover.

With standard theories handled by ground decision procedures, the next candidate for automation is quantifier reasoning. Traditional methods for first-order reasoning, such as resolution, do not extend well to the presence of decided ground theories, and therefore find little application in formal methods. (Also, formal methods often use higher-order quantification.) Fortunately, as noted above, there is generally little nesting or alternation of quantifiers in these applications, so that a combination of specialized and heuristic methods work quite well for the majority of cases (difficult cases then require user guidance). Specialized methods include those for conditional rewriting in the presence of decided theories—the close integration of rewriting with linear arithmetic is the source for much of the effectiveness of Boyer and Moore's provers [3], and similar capabilities are required in any system intended to support formal methods. Matching techniques similar to those used in rewriting can also provide heuristic instantiation for general formulas. However, my experience with PVS is that while its conditional rewriter is almost completely effective (i.e., it rarely fails to find a match if one exists), its heuristic instantiation of lemmas and general quantifier reasoning fails (usually by finding an unproductive match) more often than I would like. More effective methods for quantifier reasoning in these contexts (and for restricted instances of the higher-order case) would be a good topic for research.

Inspection of the formulas for clock synchronization shown earlier suggests that, in addition to arithmetic, propositional, and quantifier reasoning, we will also need induction. Proof that the algorithm maintains the clock synchronization conditions is accomplished using simple induction on the frame index i. Several results on finite summations are also used (a key step in the proof is to split the summation in the definition of $\Delta_p^{(i)}$ into m terms constrained by Lemma 5, and $n-m$ constrained by Lemma 4), and these require bounded induction (i.e., induction over a subrange of the natural numbers) on the recursive function that is used to define summation. Given the need for induction, it might seem that powerful automation for inductive proofs, as provided in several systems, would be beneficial. Unfortunately, these methods have generally been developed for rather restricted (e.g., equational or unquantified) logics, and not for the richer context found here. In the absence of suitable automation, the user

may be expected to indicate when induction should be used, and to identify the induction variable or expression (PVS, for example, requires this). It is then relatively straightforward to automate selection and instantiation of the appropriate induction scheme; simple tactics can finish the proof of straightforward lemmas (e.g., those needed here for properties of summations), while more explicit user guidance is be needed in more complex cases (e.g., the main induction here). Many formal methods applications require only a couple of inductions and these simple methods are adequate in these cases. Nonetheless, more automated methods (including those for generalization) would be welcome, and the development of suitable techniques is a good research topic.

3.1 Model Checking

Compared to theorem proving methods, model checking and related techniques (such as state exploration and language inclusion) are becoming rather widely used in formal methods. However, I believe that these techniques currently tend to be used standalone in application domains (such as hardware and protocols) to which they are particularly well-suited, rather than being incorporated into traditional formal methods, or integrated with theorem proving.

For my next example, I describe an experiment undertaken by my colleagues Klaus Havelund and Shankar [7], who applied a combination of finite state exploration, theorem proving, and model checking approaches to a simple protocol. Many larger and more significant problems than this have been examined by finite state enumeration and model checking techniques; what is interesting in this exercise is that it points towards an integration of these techniques with theorem proving, and also highlights some of the areas where further research is needed.

Havelund and Shankar began by reducing the protocol to finite state (by manually assigning explicit small integers as the upper bound on the size of certain data structures) and checking certain safety properties with the Murϕ explicit state exploration system [5]. They next verified these properties for the full protocol by theorem proving in PVS using a traditional invariance argument, but found in the process that the desired invariant had to be strengthened by the addition of many additional conjuncts. These were discovered incrementally during the proof attempt; each new proposed conjunct was checked with Murϕ, added to the invariant, and the evolving proof attempted once more. The whole process was iterated until a sufficiently strong invariant was developed; this eventually comprised 57 conjuncts. Seeking a better approach, they developed a finite-state abstraction of the original protocol, verified (by theorem proving) that it was indeed an abstraction, and then verified properties of the abstraction by model checking.

First, notice that the initial "reduction" to finite state in preparation for examination with Murϕ was a manual and ad-hoc process. This seems typical of finite-state analyses: the original problem is transformed by hand into a form that is acceptable to the available tool. The transformation is usually an aggressive simplification that is adequate for refutation but not for verification—meaning

that bugs found in the transformed description are likely to correspond to bugs in the original, but the failure to detect bugs in the former cannot be interpreted as verification of the latter. In the case of the protocol studied in these experiments, the maximum number of messages in a file was arbitrarily set to three: bugs that are manifest only with larger file sizes will not be found by this method.

Next, the direct verification of the full protocol was extremely tedious, as the desired safety property had to be strengthened iteratively until it became an invariant. This process took many weeks, which is clearly unacceptable for general practice. Methods for the systematic—and preferably automated—development of invariants therefore constitute a very worthwhile research topic. Of course, one of the advantages of model checking is that it is largely automatic and does not require the development of such invariants. However, when model checking is used for verification rather than refutation, it is necessary to prove that the finite-state description is a true abstraction of the original specification, and this abstraction proof may itself require invariants. Havelund and Shankar in fact reused 45 of the 57 invariants developed for their protocol in their abstraction proof, so the overall saving in effort was not great in this case. This experience highlights another very fruitful area for research: systematic and automated methods for developing finite-state abstractions. Good results are already known for some special cases [2] and I speculate that integration of these methods with model checking will eventually provide an efficient way to verify properties of infinite-state systems.

There were interesting differences between the "reduced" finite-state description checked with Murϕ and the "abstracted" version that was model checked. In the reduced Murϕ description, a file could comprise 1, 2, or 3 messages; in the abstracted description, the size of the untransmitted portion of the file is chosen from the uninterpreted enumeration `NONE`, `ONE`, and `MANY`. The relation between these different approaches—fixing the size vs. introducing abstraction (and additional nondeterminism)—is worthy of investigation.

Although these experiments indicate several areas where additional research is needed, they also demonstrate some promising directions. First, use of Murϕ to check the plausibility of proposed new invariants is representative of a useful general technique: testing conjectures using some lightweight technique before undertaking a full proof. In formal methods applications, many conjectures are false when first proposed and it is best to discover these falsehoods as early and as cheaply as possible, reserving the investment in a full proof until some confidence has been developed that it is likely to be successful. Lightweight methods generally apply to specific, or reduced, cases of the full specification, and automated assistance for creating these reduced cases is a useful addition to any support environment for formal methods. Apart from finite state enumeration, other lightweight techniques include direct evaluation (for executable specifications), and interactive simulation (for specifications that are not directly executable). The latter methods are usually based on specialized and optimized techniques for automated deduction (e.g., rewriting and enumeration over finite quantifiers).

Second, the combination of theorem proving and model checking in the last of the exercises reported above is representative of a promising direction for integrating powerful, but narrow, techniques into a larger system. For example, model checking in PVS is accomplished using an external decision procedure for Park's μ-calculus. This is extended to a decision procedure for μ-calculus on the hereditarily finite fragment of PVS's type system[6] by encoding their values in propositional variables. The branching time temporal logic CTL is then defined in PVS and its model checking problem is cast as a decision problem in μ-calculus. This allows CTL model checking to be smoothly integrated as a proof procedure in PVS. A benefit of this integration is that model checking is available for any conjecture that has the appropriate semantic attributes, independently of its linguistic representation. For example, a tabular specification construct was recently added to PVS; this was then used to formalize a requirements methodology known as SCR, and model checking was then immediately available for SCR specifications [11].

Interesting challenges for the future are to integrate other highly efficient but narrow procedures into a general purpose framework. Examples include model checking methods for hybrid systems and binary moment diagrams.

4 Interaction with the User

I believe that formal methods can deliver most value when applied to problems where traditional methods are inadequate. All the evidence points two principal sources of failure in complex systems: inadequate understanding of potential interactions, and the intrinsically hard parts of a design. Examples of the former often arise in requirements specification, where it is particularly difficult to anticipate all the interactions among the components of a system and between a system and its environment, particularly when operating in the presence of faults. In the case of Jet-Select, for example, our formalization revealed that certain interactions between error reporting and optimization allowed the possibility of firing a failed jet [6]. Examples of the latter often concern algorithms for concurrent, real time, or fault-tolerant behavior (e.g., cache-coherence or clock-synchronization)—where, again, it is difficult to anticipate all possible interactions—or highly optimized calculations whose correctness rests on a long or complex argument (e.g., SRT division and other efficient floating point algorithms).

A consequence of this observation is that automated deduction in support of formal methods will often be applied to very hard problems. It is, in my view, quite unrealistic to expect that such difficult problems can be solved automatically. The issue, then, is how should the user guide and interact with the process of automated deduction? This raises a dual issue: what information and services can automated deduction provide to the user that will assist in the analysis of very difficult problems?

[6] That is, types built recursively from the Booleans, enumerations, explicit finite subranges of the integers, and records, tuples, predicates, and functions of these.

All interaction between the user and tools for automated deduction can be considered an iteration of the following basic steps. What differs from tool to tool is the relative effort devoted to each step, and the rate of iteration.

1. Decide the procedure to be used at the next step. This can range from coarse decisions of overall strategy ("I'll use SMV") to fine issues of tactics ("instantiate the third variable of formula 3 with the following expression").
2. Transform the current representation of the problem into one that is appropriate for the procedure chosen in the previous step. This may be a major undertaking with pencil and paper (e.g., to reduce an infinite-state protocol specification to a finite-state description in the language of SMV), or it may involve mechanized transformations (including recursive application of this whole activity).
3. Set appropriate switches and dials to tune the selected procedure (e.g., choose a variable ordering for BDDs, a weighting strategy for resolution, or an ordering and orientation of lemmas for Nqthm).
4. Invoke the chosen procedure, contemplate the result returned, and iterate the whole process (sometimes, iterate locally over step 3).

My opinion is that the ability to direct this activity in an efficient and productive manner is largely determined by the predictability of the consequences selected by steps 1 and 3, the quality of information returned in step 4, and the efficiency and repeatability of step 2. The user should be able to select a procedure in step 1 on the basis of a description of what it does, not how it works. Deterministic proof procedures (e.g., elementary transformations such as a case split, or quantifier instantiation) and decision procedures are attractive from this point of view, whereas heuristic procedures are not. By the same token, the switches and dials of step 3 should be minimized, since they generally concern how a proof procedure works, rather than the substance of the conjecture under examination. Few users whose interest is formal methods are willing to learn enough about the workings of a proof procedure that they can master many choices here.

The information returned in step 4 should include the result of applying the proof procedure if it was successful (e.g., "proved," or a list of transformed or new subgoals), and an explanation if it was unsuccessful. Decision procedures and model checkers have a special value in the latter case, because they can often return a counterexample that pinpoints the source of difficulty. The ability to return useful information from failure is particularly important in applications of automated deduction to formal methods because it is to be expected that many conjectures will be false—indeed, the efficient discovery and correction of errors is one of the primary reasons for undertaking formal analysis. For this reason, techniques for automated deduction used in formal methods should not be biased towards successful outcomes—for example, they should not be set up to terminate quickly on success at the expense of taking inordinate time to discover failure.

The whole process of formal analysis will be repeated several times as errors are discovered and the design or its specification are adjusted. But the process

is not over once we successfully get to "proved" for the first time. Mechanization allows formal methods to be used to explore and refine designs—just as computational fluid dynamics is used to refine aerofoils. Our verification of clock synchronization, for example, has been modified many times: to improve the proof, to eliminate assumptions, to change the specification so that it connects better with the formalization of another part of the overall fault tolerant architecture, to tighten the bound on synchronization achieved, and to change from a Byzantine fault model to a more complex "hybrid" model [13].

The fact that formal analysis will be repeated many times as a specification is first debugged and then refined has consequences for automated deduction. First, it makes it essential, in my view, that step 2 of the interaction loop described above be automated: as the design and its specification evolve, we should recalculate the "reduced" form required for a particular proof procedure, rather than tinker with the existing one. In particular, for reliability as well as efficiency, I believe that reductions and abstractions from infinite-state to finite-state models should be formalized and mechanized, rather than left as an ad-hoc manual process. Second, the "script" of a proof needs to be recorded in manner that is reasonably robust to small changes in the specification. This argues against conducting and recording proofs in low-level and highly specific terms (e.g., "instantiate formula 3 with `x!1`" where `x!1` is the name of a Skolem constant), since the details may change with the specification. It will be more robust to indicate a procedure (e.g., "use unification to find an instantiation"), or to invoke truly automated deduction (e.g., "finish off the proof using resolution"). Finally, it is important to record dependencies among proofs and specifications, so that the user can speedily answer questions such as "what assumptions does this proof depend on?" and "what proofs may be affected if I change this lemma?"

5 Conclusion: The Need for Integration

The field of automated deduction has developed many powerful techniques that could be applied to formal methods. However, the special character of formal methods applications means that some techniques may need to be adapted to the needs of those applications, (e.g., to return more useful information on failure) and that priorities may be different than in other areas (e.g., decision procedures become more important and first order methods such as resolution may become less so). More importantly, most techniques in automated deduction, and also those related to model checking, tend to be rather brittle "point solutions" that are effective against specific classes of problems, whereas formal methods requires an integrated capability that is effective across a wide range of applications. The research challenge in this area is therefore broadly that of integration: different techniques must work together, different theories must be decided in combination, theorem proving and model checking must cooperate, and the needs and capabilities of efficient automated deduction must influence, and be influenced by, the design of expressive specification languages. Success in this endeavor will enrich both fields, providing a new and exciting application for

automated deduction, and making formal methods a truly useful and practical tool for the analysis of interesting real systems.

Acknowledgments

My opinions have formed through many stimulating discussions with my colleagues Judy Crow, David Cyrluk, Klaus Havelund, Friedrich von Henke, Patrick Lincoln, Sam Owre, N. Shankar, and M.K. Srivas, and by experiences using PVS (primarily built by Sam Owre and Shankar) and its predecessors.

References

Papers by SRI authors can generally be retrieved from `http://www.csl.sri.com/fm.html`.

[1] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[2] Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [1], pages 323–335.

[3] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. In *Machine Intelligence*, volume 11. Oxford University Press, 1986.

[4] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, number 1104 in Lecture Notes in Artificial Intelligence, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[5] David L. Dill. The Murϕ verification system. In Alur and Henzinger [1], pages 390–393.

[6] David Hamilton, Rick Covington, and John Kelly. Experiences in applying formal methods to the analysis of software and system requirements. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques*, pages 30–43, Boca Raton, FL, 1995. IEEE Computer Society.

[7] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, number 1051 in Lecture Notes in Computer Science, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.

[8] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A formal Development Support System.* Springer-Verlag, London, UK, 1991.

[9] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[10] Mogens Nielsen, Klaus Havelund, Kim Ritter Wagner, and Chris George. The RAISE language, method and tools. *Formal Aspects of Computing*, 1(1):85–114, January–March 1989.

[11] Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Available, with specification files, from `http://www.csl.sri.com/csl-95-12.html`.

[12] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[13] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Los Angeles, CA, August 1994. Association for Computing Machinery.

[14] John Rushby. Mechanizing formal methods: Opportunities and challenges. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 105–113, Limerick, Ireland, September 1995. Springer-Verlag.

[15] John Rushby and Friedrich von Henke. Formal verification of algorithms for critical systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[16] Robert E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[17] Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.

[18] Gunnar M. N. Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formula. United States Patent 5,276,897, January 4, 1994.

This article was processed using the LaTeX macro package with LLNCS style

Invited paper presented at the 16th Conference on the Foundations of Software Technology and Theoretical Computer Science, Hyderabad, India, December 1996. Springer-Verlag Lecture Notes in Computer Science, volume 1180, pages 43–51.

Mechanized Formal Methods: Progress and Prospects*

John Rushby

Computer Science Laboratory, SRI International,
Menlo Park, CA 94025, USA

Abstract. In the decade of the 1990s, formal methods have progressed from an academic curiosity at best, and a target of ridicule at worst, to a point where the leading manufacturer of microprocessors has indicated that its next design will be formally verified. In this short paper, I sketch a plausible history of the developments that led to this transformation, present a snapshot of the current state of the practice, and indicate some promising directions for the future. Mindful of the title of this conference, I suggest how formal methods might have an impact on software similar to that which they have had on hardware.

1 The Past

In their early days (the 1970s—though continuing to the present in some places), formal methods were associated with proofs of program correctness. This is not only a very costly and difficult exercise—it requires formalizing the semantics of real programming languages, and dealing with the scale and characteristics of real imperative programs—but it also adds very little value: traditional methods of code review and testing are highly effective and very few coding bugs of any significance escape detection. For example, of 197 critical faults detected during integration and system testing of the Voyager and Galileo spacecraft, just 3 were coding errors [18]. The large majority of faults arise in requirements, interfaces, and intrinsically difficult design problems (e.g., fault tolerance, and the coordination of concurrent activities). In the spacecraft data just cited, approximately 50% of faults were traced to requirements (mainly omissions), and 25% to each of interfaces and design.

During the 1980s, attention shifted from program correctness to the use of formalism in specifications, exemplified by approaches such as Z [32] and VDM [17]. Although these methods initially stressed the role of proof in development, they came to be used mainly as specification languages, and their advocates commended the utility of mathematical concepts such as sets, functions, and relations in constructing precise yet abstract descriptions of computational systems. The problem with this approach is that it is not necessary to be specifically *formal* to

* This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931.

make use of such mathematical modeling techniques; conversely, in the absence of formal proof, there are few tangible benefits to a strictly formal approach. By failing to exploit the singular characteristic of truly formal methods—namely, their ability to support deduction—specification-oriented formalisms missed the opportunity to combine mathematical modeling with calculation in the manner that has been so productive in other engineering disciplines.

The value of formal deduction is that it enables many questions about properties of formally specified requirements and designs to be settled by a systematic process that has the character of calculation. The reasons for favoring calculation over informal reasoning or trial and error experimentation are the same in computer science as in other engineering disciplines: calculation allows the properties of designs to be predicted and evaluated prior to construction, it allows analyses to be checked by others, enables large problems to be tackled in a systematic manner, and opens the door to mechanization. And in most engineering disciplines, it is mechanization that releases the full potential of mathematical modeling and calculation: the highly efficient wings of a modern airplane could not be designed without massive mechanization of computational fluid dynamics, finite element analysis, and several other branches of applied mathematics.

It was the arrival of efficient techniques for model checking in the early 1990s [19] (and related methods such as language inclusion) that first made large-scale mechanized calculations a practical reality for formal methods and demonstrated their utility to a wide audience. No less important than the techniques that made model checking practical was the change in approach and outlook that its use engendered. The limited expressiveness of the temporal logics employed in model checking means that it is seldom possible to use them to fully characterize the functionality required of a system; instead, attention is focussed on important properties that it should posses. Similarly, because model checking methods can only explore a limited, finite state space, the full system description must generally be considerably abstracted and simplified before subjecting it to model checking. Partly because of these limitations (and partly because it is able to provide excellent diagnostic information in the form of counterexamples), model checking has generally focused on *in*correctness—on finding bugs—rather than on trying to establish correctness. And find bugs it did: because model checking is well-suited to concurrent systems, it was immediately applied to some of the hardest problems in system design, such as multiprocessor cache-coherence protocols, where "high-value bugs" were quickly detected [8].

The changes in approach introduced by model checking opened up new opportunities for all formal methods: whereas previously the goal had been to specify the full functionality required, there was now seen to be a useful spectrum of desired properties; whereas previously the goal had been to describe the system in all its details, there was now seen to be value in isolating key problems and aggressively abstracting away as many details as possible; whereas previously the goal had been to establish unequivocal correctness, there was now seen to be a variety of other useful purposes that could be served by formal analysis; and whereas previously the applications had generally been to routine designs (see,

for example, the survey [9]), there was now an enthusiasm for applying formal methods to the hardest and most difficult problems of design.

Mechanized formal methods based on theorem proving, which had become modestly effective by the mid 1980s and were continually improving, benefited from the change in attitude—and the spur of competition—that came with model checking. Decision procedures for basic theories such as linear arithmetic and equality received renewed attention and acceptance, and integrated combinations of decision procedures, rewriting, and customized tactics achieved significant automation and efficiency on interesting classes of problems [22]. Most importantly, the practitioners of these approaches to formal methods followed the lead of the model checkers in applying them to complex, real-world systems [3, 35].

2 The Present

An idea of the current capabilities and accomplishments of mechanized formal methods can be obtained by considering two examples from hardware design.

The Pentium FDIV bug, which attracted a great deal of public interest, also caught the attention of the formal verification community—not least because it caused Intel to take a $475 million charge against revenues. The bug was in the lookup table of an SRT divider [25]. Binary Decision Diagrams (BDDs) have been used successfully to verify many kinds of digital circuits—but not multipliers and dividers, where they grow exponentially large [4]. Nonetheless, Bryant was able to verify a single iteration of an SRT circuit using BDDs [5]. Explosive growth of the BDD representation has generally also precluded application of symbolic model checking to dividers; however, by using a different "word level" representation, Clarke, Khaira, and Zhao were able to apply model checking to this problem [7]. Clarke, German, and Zhao were also able to verify an SRT divider using a special-purpose theorem prover based on the Mathematica symbolic algebra system [6]. Using the PVS general-purpose verification system [21], Rueß, Shankar, and Srivas gave a formally verified treatment of the general theory of SRT division, and then verified a particular circuit and lookup table [27]. While being more general, the theorem proving treatments achieved a level of automation and efficiency comparable to the BDD and model checking approaches, and were equally adept at catching errors in the tables. However, all of these treatments dealt only with the fixed-point core of the divider, and not with the issues of IEEE-compliant floating point representation. Miner and Leathrum extended the PVS treatment to include IEEE-compliance, generalized the whole development to encompass the broader class of subtractive division algorithms that includes SRT, and presented a methodology that enabled specific algorithms to be debugged and verified quite easily—which they demonstrated on various SRT tables [20].

Cache coherence protocols for distributed shared memory multiprocessors are notoriously difficult to design. Some of the early successes with symbolic model checking were in its application to this type of problem. As interest shifted from the "snoopy" to the more scalable—and much more complicated—"directory-

based" protocols, the state-explosion problem became quite severe. One response was to "downscale" (aggressively simplify) the problem, so that, for example, only two or three processors, one address, and a 1-bit data word are considered. Another was to use the various symmetries that exist in the problem to allow different, but equivalent, states to be merged. Using these and other techniques, model checkers based on both explicit state-enumeration and symbolic representations are able to tackle cache-coherence problems sufficiently well to be used in the design process for these systems [2, 11]. But although they are effective for detecting bugs, the severely downscaled models used in model checking cannot serve to verify the general case. Theorem proving techniques should be able to do this, but the difficulty of creating appropriate abstractions and sufficiently strong invariants, combined with the labor involved in guiding the theorem prover, had discouraged their application to realistic cache-coherence protocols. Recently, however, by using a method called "aggregation" to guide construction of the abstraction function, Park and Dill have been able, using PVS in a quite straightforward manner, to verify the behavior of the protocol used in the Stanford FLASH processor [24]. Furthermore, using the Murϕ explicit state-enumeration system they were able to construct an executable model for the non-sequentially-consistent memory behavior of the processor. In similar work for the Sparc V9 memory model, they were able to verify the behavior of synchronization code using Murϕ, and were able to verify the executable Murϕ model against its axiomatic specification using PVS [23].

The interesting feature of these examples is the diversity of approaches employed—and the diversity would be even greater if I had considered other examples such as pipelines, microcode, communications and switching protocols, or hybrid systems. There simply is no single best method: we are dealing with problems that are at the limit of what is computationally feasible, and different applications yield to different approaches. Thus, symbolic model checking using BDDs works well for some problems, but explicit state enumeration is better for others; some state spaces can be reduced significantly by symmetry reductions, others require partial-order reductions; some problems are best dealt with by model checking, others are better suited to theorem proving.

Just as different approaches work better for different problems, so different approaches work better for the *same* problem at different stages of its "verification lifecycle." When first encountered, a design (or its formalization) will often be full of bugs. These should be identified as quickly and as cheaply as possible. Methods that require relatively little preparation, such as typechecking, animation, or explicit state enumeration are effective here. Once the simple bugs have been eliminated, it becomes necessary to explore more and more of the state space to discover those that remain, and explicit state exploration methods that use hashing, and symbolic model checking methods, start to become more effective. Once the complete state space of downscaled instances of the problem can be explored without finding a bug, then the aggressiveness of the simplifications can be reduced, and the size of the problem instances can be increased. The "state explosion" problem is likely to hit at this point, and reduction methods

based on symmetry, partial orders, or abstraction may need to be invoked. When the largest problem instances that can be examined by finite state methods no longer reveal bugs, then it is time to consider theorem proving. For concurrent systems, it is generally necessary to develop abstractions and to strengthen the desired invariant to obtain one that is inductive. Special-purpose tools can help with these activities, and finite-state methods can be invoked during the proof process to check that proposed invariants really are so, and that subgoals are true (on finite instances) [12].

Different methods come into play on a single problem as easy bugs are eliminated and those that remain become harder to find; in a related progression, different methods come into play in the treatment of *classes* of problems as our understanding and techniques improve. For example, as enumerated above, treatments of SRT division evolved from BDD-based analysis of individual iterations, to treatment of the core of a specific algorithm by special-purpose theorem proving, to general treatment of the entire class of algorithms with a general-purpose theorem prover.

3 The Future

I offer some suggestions on likely, or promising, directions for future developments in mechanized formal methods under two headings: applications to software, and tools.

3.1 Applications to Software

Compared to hardware, software is more of a challenge for successful application of mechanized formal methods. Hardware has a relatively small number of stereotypical problems (pipeline control, floating point ALUs, microcode, cache coherence), so that the cost of developing really effective solutions can be recouped over many applications, whereas software has a vastly larger supply of problems and a correspondingly smaller community of interest for any one of them. Nonetheless, we can adopt some of the strategies that seem to have been successful for hardware.

- *Go where the bugs are.* Formal methods have been effective for hardware because their use has been targeted at areas where they can offer a real payoff: areas that experience has shown to be error-prone and where other methods are ineffective. The targeted areas concern some of the *hardest* challenges in design (e.g., the stereotypical problems mentioned above). For software, correspondingly difficult and worthwhile challenges include those where local design decisions have complex global consequences, such as the fault-tolerance and real-time properties of concurrent distributed systems, and those where independently designed systems interact, such as the problems of feature interactions, protocol stacks, and component interfaces. It is generally most productive to examine these issues at the level of the algorithms concerned, rather than at the detailed design or coding level. It

also helps to target applications where the costs of bugs are unacceptably high. These include applications that share with hardware the characteristic that design errors cannot be repaired in the field (e.g., embedded systems in consumer products), and those where failure is intolerable (e.g., safety and other kinds of critical systems).

- *Target the early lifecycle.* The requirements for hardware (especially processors) are quite simple (i.e., "implement a given instruction set architecture"), whereas those for software are generally complex (e.g., "control air traffic") and subject to change. The most damaging and costly errors discovered late in the software development lifecycle can usually be traced back to faulty requirements. Consequently, requirements validation consumes considerable resources (in avionics, for example, more than half the development costs can go into requirements; programming, in contrast, consumes less than 10%). Formal methods are singularly well-adapted to the specification and analysis of requirements, because they allow precision without premature detail (unlike pseudocode and prototyping), and they allow useful analyses to be performed on very abstract or incomplete descriptions [29].
- *Use powerful tools, and a spectrum of methods.* Without tools, formal methods are just documentation; it is tools that make formal methods useful, and powerful tools that make them productive. Many of the tools that have been effective in applications of formal methods to hardware can also be used for software (see, for example, [33], where the SMV model checker is applied to a software requirements specification); alternatively, *ideas* from those tools can be incorporated into new tools that are specifically tailored to the characteristics of software [16]. Even less than for hardware, no single tool or method provides universally effective support for all the diverse applications of formal methods to software, so a spectrum of tools and methods should be employed.

3.2 Tool Building

As noted several times already, most applications of mechanized formal methods require a range of capabilities and make use of a number of tools. Rather than loose integration of a number of different tools, however, what is really required is tight integration of a number of different capabilities [28, 31]. For example, loose integration of a theorem prover and a model checker might allow one to use a single specification of a problem and to examine specific instances with the model checker and to prove the general case with the theorem prover, whereas tight integration might allow the theorem prover actively to *use* the model checker—so that the theorem prover could set up the induction to prove the general case, with the base case and inductive step then being discharged by model checking [30]. Such an integration of theorem proving and model checking has been achieved [26], but it required extending the implementation of a complex verification system. Future systems should be designed in a much more "open" manner, so that components can be added, modified, interconnected, and accessed in a modular fashion. For example, an attractive application of formal

reasoning to software requirements is to check consistency and completeness of the conditions that label the rows and columns of tabular specifications [15]. Depending on the logic and theories used in specifying these conditions, the deductive capabilities needed to perform the checks range from propositional tautology checking, though decision procedures for ground linear arithmetic, to full interactive theorem proving. When tautology checking proved inadequate for an example derived from the TCAS II specification [13], Czerny and Heimdahl turned to the PVS verification system in order to make use of its decision procedures. However, because those decision procedures could not be accessed separately, they had to invoke the entire PVS system, which entailed more baggage and less performance than they desired [14]. What is really needed is an open environment that provides access to components such as decision procedures and the other building blocks of theorem provers and model checkers, and in which customized combinations can be quickly constructed. The hub of such an environment must be a theorem prover, since that is what has the capability to check that problems are decomposed appropriately, that constraints on the application of certain procedures are satisfied, and that all the pieces come together to solve the whole problem in a sound manner [10]. In collaboration with David Dill of Stanford University, we are about to begin construction of such an environment.

4 Conclusion

These are exciting times for mechanized formal methods, with opportunities for rapid and significant progress in the capabilities of tools and the quality and scale of their applications. Theoretical research can assist these developments by, for example, providing better characterizations for the complexities of the various problems and algorithms encountered (almost every problem in model checking and theorem proving is NP-hard or worse), and by identifying useful special cases that admit fast solutions.

References

Papers by SRI authors are generally available from `http://www.csl.sri.com/fm.html`.

[1] Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[2] Ásgeir Th. Eiríksson and Ken L. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 367–380, Liege, Belgium, June 1995. Springer-Verlag.

[3] Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Srivas and Camilleri [34], pages 275–293.

[4] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[5] Randal E. Bryant. Bit-level analysis of an SRT divider circuit. In *Proceedings of the 33rd Design Automation Conference*, pages 661–665, Las Vegas, NV, June 1996.

[6] E. M. Clarke, S. M. German, and X. Zhao. Verifying the SRT division algorithm using theorem proving techniques. In Alur and Henzinger [1], pages 111–122.

[7] E. M. Clarke, Manpreet Khaira, and Xudong Zhao. Word level symbolic model checking—avoiding the Pentium FDIV error. In *Proceedings of the 33rd Design Automation Conference*, pages 645–648, Las Veqas, NV, June 1996.

[8] Edmund M. Clarke, Orna Grumberg, Hiromi Haraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2):217–232, March 1995.

[9] Dan Craigen, Susan Gerhart, and Ted Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.

[10] David A. Cyrluk and Mandayam K. Srivas. Theorem proving: Not an esoteric diversion, but the unifying framework for industrial verification. In *International Conference on Computer Design: VLSI in Computers and Processors (ICCD '95)*, pages 538–544, Austin, TX, October 1995. IEEE Computer Society.

[11] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, October 1992. Cambridge, MA.

[12] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, volume 1051 of *Lecture Notes in Computer Science*, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.

[13] Mats P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In Steven J. Zeil, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 79–83, San Diego, CA, January 1996. Association for Computing Machinery.

[14] Mats P. E. Heimdahl and Barbara J. Czerny. Using PVS to analyze hierarchical state-based requirements for completeness and consistency. In *IEEE High-Assurance Systems Engineering Workshop (HASE '96)*, Niagara on the Lake, Canada, October 1996.

[15] Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[16] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[17] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science, Hemel Hempstead, UK, 1990.

[18] Robyn R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *IEEE International Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, January 1993.

[19] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, MA, 1993.

[20] Paul S. Miner and James F. Leathrum, Jr. Verification of IEEE compliant subtractive division algorithms. In Srivas and Camilleri [34], pages 64–78.

[21] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Alur and Henzinger [1], pages 411–414.

[22] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[23] Seungjoon Park and David L. Dill. An executable specification, analyzer and verifier for RMO (Relaxed Memory Order). In *7th ACM Symposium on Parallel Algorithms and Architectures*, pages 34–51, July 1995.

[24] Seungjoon Park and David L. Dill. Verification of the FLASH cache coherence protocol by aggregation of distributed transactions. In *8th ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, Padua, Italy, June 1996.

[25] Vaughan Pratt. Anatomy of the Pentium bug. In *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107, Aarhus, Denmark, May 1995. Springer-Verlag.

[26] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[27] H. Rueß, N. Shankar, and M. K. Srivas. Modular verification of SRT division. In Alur and Henzinger [1], pages 123–134.

[28] John Rushby. Automated deduction and formal methods. In Alur and Henzinger [1], pages 169–183.

[29] John Rushby. Calculating with requirements. In *3rd IEEE International Symposium on Requirements Engineering*, Annapolis, MD, January 1997. IEEE Computer Society. To appear.

[30] N. Shankar. Computer-aided computing. In Bernhard Möller, editor, *Mathematics of Program Construction '95*, volume 947 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1995.

[31] Natarajan Shankar. Unifying verification paradigms. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*, pages 22–39, Uppsala, Sweden, September 1996. Springer-Verlag.

[32] J. M. Spivey, editor. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, Hemel Hempstead, UK, 1993.

[33] Tirumale Sreemani and Joanne M. Atlee. Feasibility of model checking software requirements. In *COMPASS '96 (Proceedings of the Eleventh Annual Conference on Computer Assurance)*, pages 77–88, Gaithersburg, MD, June 1996. IEEE Washington Section.

[34] Mandayam Srivas and Albert Camilleri, editors. *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1996. Springer-Verlag.

[35] Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

This article was processed using the LaTeX macro package with LLNCS style

Invited paper presented at the 3rd IEEE International Symposium on Requirements Engineering, Annapolis, MD, January, 1997. IEEE Computer Society Press, pp. 144–146.

Calculating with Requirements*

(Extended Abstract)

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

1 Automated Formal Methods

Requirements *elicitation* is concerned with discovering what is wanted; it necessarily depends on social processes such as discussion, introspection, review of documentation, and experimentation. Requirements *engineering*, on the other hand, is concerned with turning the products of elicitation into precise, unambiguous, and complete descriptions of what the system under consideration is to do. Although they shade into and complement each other, and may be part of a larger iterative process, requirements elicitation and engineering are different activities that need to be supported by different techniques and tools.

I maintain that *formal methods* provide techniques and tools that are appropriate—and effective—for the requirements engineering activity. They are effective because they allow certain questions about requirements to be reduced to calculations, and this is valuable because it allows *reviews* to be supplemented or replaced by *analyses*. I am using these terms in the sense in which they are employed in the guidelines for software on commercial aircraft [10, Section 6.3]: reviews are processes that depend on human judgment and consensus, while analyses are objective "mechanical" processes such as testing or calculation. Of course, certain questions do require human judgment, and some decisions require consensus, but many other issues are better addressed by analyses than by reviews: analyses are systematic, can be checked by others, and can even be automated. Especially when automated, analyses can be more reliable and thorough than reviews, and cheaper.

Formal methods are often advocated for the intellectual framework that they provide, and the methodological benefits that are believed to accompany their use. While I certainly agree that formal methods can help designers to conceptualize and formulate issues and requirements in a perspicuous and productive manner, I am skeptical that assurance based on human review is any more reliable for formal specifications than for informal descriptions. Instead, I claim that the distinctive benefit of specifically *formal* methods is that they support analysis through formal deduction—that is by theorem proving and related methods such as model checking. These are systematic processes that have the character of calculation and, like numerical calculations, they can be automated. The reasons for favoring mathematical modeling and calculation are the same in computer science as in other engineering disciplines: they allow the consequences of requirements and the properties of designs to be accurately predicted and evaluated prior to construction.

In most engineering disciplines, it is automation that releases the full potential of mathematical modeling: the highly efficient wings of a modern airplane could not be designed without the massive automation of computational fluid dynamics, finite element analysis, and several other mathematical modeling techniques. Automating the calculations underlying formal methods yields similar benefits for computer science: many questions that are currently examined by intensive but unreliable reviews, or by massive but necessarily incomplete testing, can be settled by automated calculation—thereby providing analysis that is more reliable and more comprehensive than at present, and liberating human reviewers for more creative and challenging tasks that truly require their judgment.

2 Experiments

Over the last few years, application of specialized but pragmatically effective theorem proving techniques, and of model checking and related methods, has made it possible to subject formal requirements specifications to several kinds of automated analysis. Some of these have been applied experimentally to require-

*This work was partially supported by NASA Langley Research Center under contract NAS1-20334, by the Air Force Office of Scientific research, Air Force Materiel Command, USAF, under contract F49620-95-C0044, and by the National Science Foundation under contract CCR-9509931.

ments documented in "Change Requests" (CRs) for the flight software of the Space Shuttle by a team involving the group at Lockheed Martin (formerly IBM) that develops this software, several NASA centers (Johnson, Langley, and JPL), and SRI. These experiments, running alongside what is generally considered an exemplary process for requirements review, provide useful anecdotal evidence for the effectiveness of automated formal analyses.

2.1 Very Strong Typechecking

One of the most basic "theorems" that can be proved about a specification is type-correctness. For programming languages, this theorem is "proved" by simple typechecking algorithms, but specification languages can use general theorem proving for this purpose with a corresponding increase in the strictness in the notion of type-correctness that can be enforced [9]. By simply describing its interfaces and functional requirements in such a very strongly-typed specification language, 86 issues (including 11 considered "major") were detected in several iterations of a CR for installing Global Positioning System (GPS) navigation on the Shuttle [4].[1]

2.2 Completeness and Consistency of Tabular Specifications

Rather stronger than typechecking are completeness and consistency checks for tabular specifications of the kind advocated by Parnas [14]. The idea is that the rows and columns of the table should partition the input space into distinct regions; if the circumstance in which each column (*resp.* row) applies is specified by a logical formula, then consistency requires that the pairwise conjunctions of the formulas should be false, and completeness requires that their overall disjunction should be true. Depending on the logic and theories used in the formulas, the deductive capabilities needed to discharge these proof obligations range from propositional tautology checking, though decision procedures for ground linear arithmetic, to full interactive theorem proving [11]. The capabilities at the lower end of this spectrum can be automated very effectively and can scale up to very large tables; those at the upper end are less automated and scale less well. Rapid progress is being made on developing balanced techniques that combine efficient and scalable automation with adequate expressiveness [1]. Applying some of these methods to a Shuttle CR for the "Heading Alignment Cylinder" (HAC), revealed that several rows in one table had overlapping conditions, and that several unstated assumptions needed to be articulated before the completeness and consistency of others could be established [3]. The Shuttle requirements analysts considered discovery of the missing assumptions to be particularly valuable. Similar successes with automated completeness and consistency checking have been reported for the RSML [6] and SCR [7] requirements methods.

2.3 State Exploration and Model Checking

Deeper analysis of tabular specifications is possible using model checking to examine whether certain desired properties are invariant or reachable. Atlee and Sreemani have demonstrated that such methods scale to quite large examples [13]. Model checking can also be used to perform analyses that are intermediate between testing individual cases (as can be performed by prototyping or animation), and proving general theorems. The idea is to "downscale" (aggressively simplify) the description of the system and its environment so that they become finite state. Finite state exploration or model checking methods can then be used to examine all the reachable states. Although the necessary simplification may be too aggressive for verification (i.e., properties of the simplified model may not be true of the full system) it is often adequate for refutation (i.e., bugs found in the model will also exist in the full system). Anecdotally, it seems that exploring *all* the states of a simplified system specification is much more effective at finding bugs than sampling those of the full system by prototyping or animation. These techniques are applied routinely (and very effectively) to hardware and protocols, but their application to software and to requirements is relatively new (see, for example [8]). Applying them to a Shuttle CR for three-engines-out abort sequencing revealed 19 errors, of which only 1 (albeit the most significant) had been discovered by human review [2].

2.4 Formal Challenges

Some of the most searching examinations of requirements specifications can be performed by "challenging" the specification with a putative theorem: "if this specification captures my intent, then the following ought to follow." Suppose, for example, that we had specified the operation of sorting a sequence; we might then challenge the specification by asking whether sorting an already sorted sequence leaves the

[1] At an earlier stage in this activity, formal methods had detected 30 issues, including 7 considered major, compared with 4, of which 1 was considered major, for the human review-based process [2].

sequence unchanged (i.e., we attempt to prove the theorem $\forall s : sort(sort(s)) = sort(s)$). Such a challenge would reveal a deficiency in many of the early specifications of sorting, which required the output to be ordered, but neglected to stipulate that it should also be a permutation of the input [5]. Applying this approach to the mature requirements specification for a Shuttle function known as "Jet Select" suggested the challenge "no jet that is designated as failed shall be selected for firing" [12]. The attempt to prove this was unsuccessful, revealing thereby a significant and previously undiscovered problem with the existing requirements specification.

3 Summary

Many issues in requirements engineering can be explored and analyzed using automated formal methods. At present, the tools supporting these analyses are not ideal: considerable knowledge and experience are required to select the most appropriate tool for a given task, to formulate the problem in a suitable manner, and to coax the tool into divulging a useful result. However, engineering challenges in the integration and scaling of formal methods tools are being rapidly overcome and these difficulties should soon be significantly reduced. Judicious use of automated formal calculations will allow many issues to be explored more completely and reliably than at present, and will allow the talents of human reviewers to be reserved for those problems in requirements analysis that truly require them.

References

Papers by SRI authors can generally be retrieved from `http://www.csl.sri.com/fm.html`.

[1] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201, Palo Alto, CA, November 1996. Springer-Verlag.

[2] Judith Crow and Ben L. Di Vito. Formalizing Space Shuttle software requirements. In *First Workshop on Formal Methods in Software Practice (FMSP '96)*, pages 40–48, San Diego, CA, January 1996. Association for Computing Machinery.

[3] Judith Crow and Ben L. Di Vito. Formalizing space shuttle software requirements: Four case studies. Submitted for publication, 1997.

[4] Ben L. Di Vito. Formalizing new navigation requirements for NASA's space shuttle. In *Formal Methods Europe FME '96*, volume 1051 of *Lecture Notes in Computer Science*, pages 160–178, Oxford, UK, March 1996. Springer-Verlag.

[5] S. L. Gerhart and L. Yelowitz. Observations of fallibility in modern programming methodologies. *IEEE Transactions on Software Engineering*, SE-2(3):195–207, September 1976.

[6] Mats P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In Steven J. Zeil, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 79–83, San Diego, CA, January 1996. Association for Computing Machinery.

[7] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In *COMPASS '95 (Proceedings of the Tenth Annual Conference on Computer Assurance)*, pages 109–122, Gaithersburg, MD, June 1995. IEEE Washington Section.

[8] Daniel Jackson and Craig A. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7):484–495, July 1996.

[9] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[10] *DO-178B: Software Considerations in Airborne Systems and Equipment Certification.* Requirements and Technical Concepts for Aviation, Washington, DC, December 1992. This document is known as EUROCAE ED-12B in Europe.

[11] John Rushby. Mechanizing formal methods: Opportunities and challenges. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM '95: The Z Formal Specification Notation; 9th International Conference of Z Users*, volume 967 of *Lecture Notes in Computer Science*, pages 105–113, Limerick, Ireland, September 1995. Springer-Verlag.

[12] John Rushby. Automated deduction and formal methods. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 169–183, New Brunswick, NJ, Jul/Aug 1996. Springer-Verlag.

[13] Tirumale Sreemani and Joanne M. Atlee. Feasibility of model checking software requirements. In *COMPASS '96 (Proceedings of the Eleventh Annual Conference on Computer Assurance)*, pages 77–88, Gaithersburg, MD, June 1996. IEEE Washington Section.

[14] A. John van Schouwen, David Lorge Parnas, and Jan Madey. Documentation of requirements for computer systems. In *IEEE International Symposium on Requirements Engineering*, pages 198–207, San Diego, CA, January 1993.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Unifying Verification Paradigms (Extended Abstract)*

To appear in the Proceedings of FTRTFT'96

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (415) 859-5272 Fax: +1 (415) 859-2844

Abstract. The field of formal methods is blessed with an overabundance of formalisms (functional, relational, automata-theoretic, modal, and temporal), techniques (resolution, rewriting, induction, and model checking), and application areas (hardware, reactive, fault-tolerant, real-time, and hybrid systems). No single verification approach has proven convincingly superior to the others. I argue that it is both necessary and desirable to develop a unified framework within which different approaches can coexist. The paper outlines some preliminary efforts in this direction in the context of SRI's PVS system. These efforts include the embedding of special-purpose formalisms (e.g., the Duration Calculus) into the general-purpose PVS logic, the integration of theorem proving with various forms of model checking, and the application of theorem proving and model checking to the analysis of tabular specifications.

1 Introduction

There has been a dramatic proliferation of verification formalisms in recent times. Volume B of the *Handbook of Theoretical Computer Science* [40] takes up the bulk of nearly 1300 pages to survey a small fraction of these formalisms (e.g.,

* Supported by the Air Force Office of Scientific Research under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931 and CCR-930044. Some of the applications described were undertaken for NASA Langley Research Center under contracts NAS1-18969 and NAS1-20334 and for ARPA through NASA Ames Research Center under contract NASA-NAG-2-891. The views presented in this paper bear the influence of several of my colleagues including John Rushby, Sam Owre, Jens Skakkebæk, Judy Crow, David Cyrluk, Mandayam Srivas, Patrick Lincoln, and Klaus Havelund. Whether they or the funding agencies would concur with these views is another matter. The word *paradigm* is perhaps being misused in the title, but the author is not the first to perpetrate this abuse.

Hoare logics, modal and temporal logics, process algebras). Clearly, many of these formalisms are useful in obtaining a metatheoretic characterization of completeness, expressiveness, and decidability, but from a practical viewpoint, one has to wonder whether this cornucopia of formalisms really offers any significant advantage over conventional mathematics (e.g., predicate calculus, set theory, algebra, analysis, combinatorics). This multiplicity of formalisms raises several obvious questions and challenges:

- Who needs so many formalisms?
- Does any one formalism address a practical problem in its totality?
- How can we make it easier to implement a novel idea or formalism?
- How can we make different formalisms interact in a coherent manner?

In response to these questions, I believe that:

- Though ordinary mathematics is adequate for describing computational phenomena, formalism provides a convenient layer of abstraction along with notational and deductive tools. For example, temporal logic [28] can succinctly capture the fairness assumptions and the safety and liveness properties of finite-state programs. Furthermore, the propositional fragment of temporal logic is decidable. This fragment of temporal logic can also be used for model checking and for synthesizing control programs [6]. Similar arguments can be made for the numerous variants of Hoare logics, process algebras, algebraic datatype specification languages, dataflow languages, and others.
- In my experience, no individual special-purpose formalism fully addresses the verification problems of practical significance. Though formalism does make it convenient to use mathematics, it does not avoid the need for mathematics. For many practical problems, the main burden of verification is still in the sheer amount of conventional mathematics required to carry out a successful verification. For example, Dutertre [8] found it necessary to build a library of basic facts of mathematical analysis as a prelude to the formalization of hybrid systems. Other work in the verification of real-time and fault-tolerant systems [7] also exhibits a heavy dependence on basic mathematics that are not alleviated by the use of special-purpose formalisms.
- Logical frameworks like Isabelle [33] do make it easier to implement proof checkers for new formalisms. However, as we have already seen, individual formalisms are not adequate for practical verification. For verification purposes, we can obtain a more synergistic interaction between different formalisms by embedding these as theories and notational front-ends within a general-purpose framework for conventional mathematics. In this paper, we describe a few such attempts at integrating special-purpose formalisms and deductive procedures into the framework of PVS.
- Even if we can embed special-purpose formalisms within a general-purpose verification framework, very little has been achieved unless these formalisms can interact smoothly. For example, a unified approach should allow the use

of induction, rewriting, and propositional logic for the verification of hardware components, and the application of temporal logic or process algebra for verifying the asynchronous interaction between hardware components.

In an attempt to do something about these beliefs, we have been engaged in the development of a verification framework named PVS [30]. While PVS is a general-purpose verification system based on typed higher-order logic, it provides a notational framework for describing computational phenomena, and incorporates a variety of decision procedures for reasoning effectively about computation. PVS has been used to:

- Develop domain libraries for finite sets, cardinalities, bit-vectors, and reals [8].[2]
- Verify challenging exercises from areas such as hardware [39], fault-tolerant algorithms [27], real-time computing, and reactive systems [7, 19].
- Embed various formalisms like the Duration Calculus [37,38], branching and linear-time modal logics [34], action systems [25], and I/O automata [1].

This paper describes some of the steps taken in PVS towards a coherent integration of different verification paradigms, and the larger challenges that remain. The basic theme of this paper is that the effective mechanization of formal methods requires a careful and coherent integration of mathematical theories, deductive procedures, and special-purpose formalisms. Section 2 outlines a small PVS example showing that a fair amount of conventional mathematics is needed to formalize and verify even elementary computational problems. We then give three examples of integration of special-purpose formalisms and deductive techniques within the general-purpose framework provided by PVS. Section 3 describes the integration of symbolic model checking as a decision procedure in PVS. It shows how temporal program logics like CTL and can be defined in PVS, and model checking and theorem proving can be combined in the verification of an N-process mutual exclusion algorithm. Section 4 describes the construction of a Duration Calculus verification tool by means of a semantic embedding in PVS. The advantage of this approach is that deductive capabilities of PVS are combined with the notational convenience of the Duration Calculus. Section 5 outlines the extension of PVS with support for various kinds of tabular specification notations that are popular in software requirements engineering. The deductive and model checking capabilities of PVS are used in the analysis of such tabular specifications. Section 6 presents some observations and challenges.

2 The Need for Conventional Mathematics

This section describes a small PVS example showing that a significant amount of conventional mathematics is required to formalize even basic computational phenomena. This example is the well known Rabin-Scott result on the equivalence

[2] The PVS libraries and numerous PVS-related papers and reports are available from the URL: `http://www.csl.sri.com/pvs.html`.

of deterministic (DFA) and nondeterministic (NFA) versions of finite automata.[3] The following specification of a DFA happens to be in PVS but it is not that different from the corresponding textbook specification. It describes a theory with parameters `Sigma` denoting the alphabet type, `state` denoting the state space type, `start` identifying a distinguished start state, a transition function `delta` mapping each alphabet-state combination to the end state of the transition, and a set[4] `final?` of accepting or final states. The iterated transition function on a string of the alphabet `Sigma` is given by `DELTA`. The set of accepted strings is characterized by the predicate `DAccept?` which checks that the state returned by `DELTA` on the given string and the start state is a final state.

```
DFA  [Sigma : TYPE,
      state : TYPE,
      start : state,
      delta : [Sigma -> [state -> state]],
      final? : set[state] ]
   : THEORY
 BEGIN

  DELTA((string : list[Sigma]))((S : state)):
            RECURSIVE state =
    (CASES string OF
        null : S,
        cons(a, x): delta(a)(DELTA(x)(S))
     ENDCASES)
    MEASURE length(string)

  DAccept?((string : list[Sigma])) : bool =
      final?(DELTA(string)(start))
 END DFA
```

In this simple specification, we have already seen the use of sets (i.e., predicates like `final?`), curried higher-order operations (`delta` and `DELTA`), abstract datatypes (`list`), recursion, and arithmetic (for the termination of `DELTA`). The purpose of the curried form of the transition function `delta` will become apparent in the specification `NFA` below. The reason for using a parametric theory for `DFA` is so that useful definitions and facts about DFAs can be imported by suitably instantiating these parameters, as is done in the theory `equiv` below.

The specification of NFAs is similar. We first define the notion of a union or least upper bound `LUB(SS)` of a set of sets `SS` over the type `T`.

[3] This example was posed as a challenge by Debora Weber-Wulff.

[4] PVS is a typed higher-order logic where sets are just predicates over a specific type, rather than the richer notion given by, say, Zermelo-Fraenkel set theory.

```
LUB [T : TYPE]: THEORY
  BEGIN

    LUB((SS: set[set[T]]))(s : T) : bool =
        (EXISTS (S : set[T]): SS(S) AND S(s))
  END LUB
```

The key difference between the NFA and DFA theories is that the range of ndelta is now a set of states corresponding to a nondeterministic transition. The iterated transition function NDELTA returns a set of states by taking the union of the image[5] of the curried operation ndelta with respect to the set of states returned in the recursive invocation of NDELTA. A string is accepted according to Accept? if and only if the set of states returned by NDELTA on the string and the start state contains a final state.

```
NFA   [Sigma : TYPE,
       state : TYPE,
       start : state,
       ndelta : [Sigma -> [state -> set[state]]],
       final? : set[state] ]
    : THEORY
  BEGIN
    IMPORTING LUB[state]

    NDELTA((string : list[Sigma]))((s : state)) :
           RECURSIVE set[state] =
       (CASES string OF
         null : singleton(s),
         cons(a, x): LUB(image(ndelta(a), NDELTA(x)(s)))
        ENDCASES)
     MEASURE length(string)

   Accept?((string : list[Sigma])) : bool =
     (EXISTS (r : (final?)) :
       member(r, NDELTA(string)(start)))
  END NFA
```

Since every DFA is trivially an NFA, the equivalence between NFAs and DFAs is shown by constructing a DFA with the same language as a given NFA. This is done in the theory equiv which has the same parameterization as the NFA theory. The first declaration imports the corresponding NFA theory.

[5] The image operation is defined in the PVS prelude theories.

```
equiv[Sigma : TYPE,
      state : TYPE,
      start : state,
      ndelta : [Sigma -> [state -> set[state]]],
      final? : set[state] ]: THEORY

 BEGIN

  IMPORTING NFA[Sigma, state, start, ndelta, final?]
  ⋮
 END equiv
```

The corresponding DFA theory is imported by first constructing a DFA state type dstate as the type consisting of subsets of the NFA state type state. The corresponding deterministic transition function delta then maps a dstate to a dstate so as to correspond to an NFA transition. The final DFA states given by dfinal? are those subsets of state that contain an NFA final state. The DFA start state is the singleton set containing the NFA start state. We can now import the appropriate instance of DFA.

```
  dstate: TYPE = set[state]

  delta((symbol : Sigma))((S : dstate)): dstate =
       LUB(image(ndelta(symbol), S))

  dfinal?((S : dstate)) : bool =
    (EXISTS (r : (final?)) : member(r, S))

  dstart : dstate = singleton(start)

  IMPORTING DFA[Sigma, dstate, dstart, delta, dfinal?]
```

The equivalence between the iterated transition functions given by lemma main can be established by a completely trivial structural induction on the string parameter. The language equivalence given by equiv then follows by straightforward predicate calculus.

```
  main: LEMMA
   (FORALL (x : list[Sigma]), (s : state):
      NDELTA(x)(s) = DELTA(x)(singleton(s)))

  equiv: THEOREM
   (FORALL (string : list[Sigma]):
      Accept?(string) IFF DAccept?(string))
```

The equivalence between NFAs with and without epsilon transitions (i.e., transitions that do not consume a symbol from the string) is correspondingly

trivial. It too involves such familiar ideas as inductive definitions, structural induction, definition expansion, and predicate calculus. The textbook proofs of this equivalence are unduly circuitous [21]. They also overlook the observation that these equivalences make no use of the finiteness of the `state` type. The main point is that a lot of basic mathematics (predicate calculus, set theory, lists, arithmetic, recursion) is needed to describe even a simple computational phenomenon. The mechanical proofs are trivial because of the tightly integrated mechanization of equational reasoning (i.e., congruence closure for propagating ground equality information and rewriting with respect to definitions and lemmas), predicate calculus, set manipulations, datatype simplifications, and structural induction over recursive datatypes.

3 Integrating Theorem Proving and Model Checking

This section descibes the embedding of the mu-calculus and the branching-time temporal logic CTL in PVS, and the integration of model checking as a deductive procedure for handle properties stated in this formalism. This integration makes it possible to apply model checking and theorem proving in conjunction to verification tasks where neither technique in isolation would be effective.

Verification by means of theorem proving establishes a property P of a system M by proving P from M, i.e., $M \vdash P$, whereas model checking [6] demonstrates that M satisfies P, i.e., $M \models P$. Typically in model checking, M is a finite Kripke model of a system consisting of states and transitions between states. The model M can be either in an *explicit* state representation as an assignment of values to variables, or in a symbolic form where sets of states are represented by a Boolean formula, and the transition relations are also represented by Boolean formulas. The property P is stated in a linear or branching time temporal logic. The model checker can then deduce whether the model M satisfies P by progressively annotating each state with subformulas of P that hold on it.

Model checking can be viewed as a decision procedure for the finite-state fragment of a more general theory of extremal fixpoints of monotone predicate transformers. This theory is based on Park's mu-calculus [32]. Emerson and Lei [9] showed how the mu-calculus can be used to embed a number of finite-state formalisms. They also gave an efficient algorithm for checking mu-calculus formulas. Burch, Clarke, Dill, McMillan, and Hwang [3] showed how Park's mu-calculus could be decided using a symbolic representation in terms of binary decision diagrams. They showed how several other model checking paradigms such as branching and linear-time model checking, LTL validity checking, simulation checking in process algebra, and language containment could be captured within the unified framework of the mu-calculus. In joint work with Rajan and Srivas [34], we showed how the mu-calculus could be defined in a higher-order logic as a way of combining the benefits of theorem proving and model checking. We give a brief outline of this integration.

Park's mu-calculus can be presented in a definitional way (i.e., without axioms) in higher order logic. The PVS theory `mucalculus` below takes a parameter `T` corrresponding to the state type of the computation system or Kripke model. We can then lift the implication ordering on formulas to predicates on states. A monotone predicate transformer is one that preserves this ordering relation as checked by `ismono`. A fixpoint of a predicate transformer is defined by the relation `isfix`.

```
mucalculus[T:TYPE]: THEORY
BEGIN
  s: VAR T
  p,p1,p2: VAR pred[T]
  predtt: TYPE = [pred[T]->pred[T]]
  pp: VAR predtt
  setofpred: VAR pred[pred[T]]

  IMPORTING orders[pred[T]]
  <=(p1,p2):bool = FORALL s: p1(s) IMPLIES p2(s)

  ismono(pp):bool =
    FORALL p1,p2: p1 <= p2 IMPLIES pp(p1) <= pp(p2)

  isfix (pp,p):bool = (pp(p) = p)
  ⋮
END mucalculus
```

The least fixpoint is defined as the greatest lower bound of the pre-fixpoints and this is used to define the μ operator.

```
  glb (setofpred):pred[T] =
    LAMBDA s: (FORALL p: member(p,setofpred) IMPLIES  p(s))

  lfp (pp):pred[T] = glb({p | pp(p) <= p})

  mu(pp): pred[T] = lfp(pp)
```

Correspondingly, we can prove that the least upper bound of the post-fixpoints of a monotone predicate transformer yields the greatest fixpoint, which then defines the ν operator.

```
  lub (setofpred):pred[T] =
    LAMBDA s: EXISTS p: member(p,setofpred) AND p(s)

  gfp (pp):pred[T] = lub({p | p <= (pp(p))})

  nu (pp):pred[T] = gfp(pp)
```

We can informally write `mu(LAMBDA Z: P(Z))` as $\mu Z.P(Z)$. The fixpoint operators can be explained intuitively in terms of infinite disjunctions as follows. Let $\top$ be $\lambda z_1 \ldots, z_n$: **true** and $\bot$ be $\lambda z_1, \ldots, z_n$: **false**, then if P is $\cup$-continuous (i.e., $P[\cup_{i\in\omega} Z_i] = \cup_{i\in\omega} P[Z_i]$, where $Z_j \leq Z_k$ for $j < k$), and Q is $\cap$-continuous (i.e., $P[\cap_{i\in\omega} Z_i] = \cap_{i\in\omega} P[Z_i]$, where $Z_j \leq Z_k$ for $k < j$), then

$$(\mu Z.P[Z])(z_1, \ldots, z_n) = \bigvee_i P^i[\bot](z_1, \ldots, z_n)$$

$$(\nu Z.Q[Z])(z_1, \ldots, z_n) = \bigwedge_i Q^i[\top](z_1, \ldots, z_n)$$

Using PVS, we can prove that the `lfp` operator does actually return the least fixpoint, and consequently derive a useful least fixpoint induction principle. We can also prove various lemmas such as that μ and ν are duals, that $\cup$- or $\cap$-continuity implies monotonicity, and that for $\cup$-continuous predicate transformers P, the least fixpoint can be obtained by taking the least upper bound of iterates of P from $\bot$. None of these results requires the computation state space to be of bounded finite size. When the state type `T` is finite, the least and greatest fixpoints can be explicitly computed. In PVS, this is done by carrying out a binary encoding the variables of state type as an n-tuple of booleans, and using a package due to Janssen [24] which employs binary decision diagrams (BDDs) to symbolically compute least and greatest fixpoints over the booleans. With the ability to represent finite states and compute fixpoints, we can, for example, compute the reachability relation on the program graph and represent the operators of the branching-time temporal logic CTL.

The temporal operators of CTL are easily defined in terms of the mu-calculus. We give just three simple instances of these definitions. These operators are indexed by the next-state relation `N` of the computation. The temporal operator `EX` is defined so that `EX(N, p)` holds of a state if `p` holds of some next state. The temporal formula `EG(N, p)` holds of a state if there is an infinite computation path leading out where `p` always holds. The temporal formula `EU(N, p, q)` holds of a state if there is an infinite outgoing computation path where q eventually holds upto which point p holds. Note that the `connectives` theory overloads `AND`, `OR`, `NOT` to operate on predicates rather than booleans.

```
ctlops[state : TYPE]: THEORY
 BEGIN
  IMPORTING mucalculus[state], connectives[state]
  u,v,w: VAR state
  f,g,Q,P,p1,p2: VAR pred[state]
  Z: VAR pred[[state, state]]
  N: VAR [state, state->bool]

  EX(N,f)(u):bool =  (EXISTS v: (f(v) AND N(u, v)))

  EG(N,f):pred[state] = nu (LAMBDA Q: (f AND EX(N,Q)))

  EU(N,f,g):pred[state] = mu (LAMBDA Q: (g OR (f AND EX(N,Q))))
  .
  .
  .
END ctlops
```

Fairness cannot be expressed in CTL but a fairness-enhanced CTL called fairCTL can be defined using the mu-calculus. Using fixpoint induction, we can prove a few useful theorems about CTL. The crucial theorem we have proved is a preservation theorem for AG properties that provides a sufficient condition for an abstraction mapping between two Kripke models to preserve the validity of AG properties. Using this theorem, we can establish the main safety property of an N-process mutual exclusion algorithm by applying a suitable abstraction to the induction step to reduce it to the correctness of a two-process protocol that is easily verified by means of model checking. Since the N-process algorithm is not a finite-state one, model checking alone will not be applicable here. Theorem proving could be used to verify the safety property, but this typically requires the discovery of a suitably strengthened invariant. The benefit of using model checking is that one does not need to discover a suitably strengthened inductive invariant. The discovery of the right abstraction mapping still requires some creativity but this is usually simpler than invariant discovery. In joint work with Havelund [12], we have applied abstraction to reduce the verification of a communication protocol to a form amenable to model checking.

With Rajan and Schuerman, we have also integrated model checking for the linear-time temporal logic LTL. Note that whereas branching-time properties are predicates on computation states, linear-time properties are predicates on computation traces. The integration of LTL model checking is done by translating LTL formulas into a Kripke automaton which is composed with the given Kripke model [5]. The LTL model checking problem then reduces to fairCTL model checking on this extended automaton. The integration of LTL is not as smooth as that of CTL since it requires a hand-crafted translation of the LTL satisfaction relation to that for fairCTL. However, the integration still retains the benefit allowing the combined use of theorem proving and model checking.

As already seen in Section 2, a fair amount of basic mathematics is needed to formalize the mu-calculus and CTL. Since the semantics of these formalisms has been embedded in PVS, we can combine the use of theorem proving and

model checking within a single framework by formally decomposing problems into control-intensive parts where model checking is effective, and data-intensive parts where theorem proving techniques like rewriting and induction are needed. Such decompositions are crucial for the successful use of model checking.

4 Embedding the Duration Calculus

This section describes the embedding of a special-purpose formalism, the Duration Calculus (DC), within PVS. This embedding results in a system PC/DC where all the resources of PVS are exploited within the notational and deductive framework of the Duration Calculus. Since DC is used for reasoning about real-time systems, the arithmetic capabilities of PVS are especially useful for verifying DC properties. The embedding here of the Duration Calculus within PVS is not dissimilar to the semantic definition of the mu-calculus and CTL given in Section 3. The crucial difference here is that the proof theory of DC is developed and used to construct a proof checker for DC specifications and properties. Gordon [11] gives a good introduction to the embedding of program logics within higher-order logic.

The Duration Calculus [4] is a real-time interval temporal logic. A DC formula is a predicate over a time interval $[b, e]$, where $b \leq e$. Apart from the usual logical connectives and quantifiers, DC formulas can be constructed using a *chop* operator so that $A; B$ holds of $[b, e]$ if for some m, $b \leq m \leq e$, A holds of $[b, m]$ and B holds of $[m, e]$. The chop operator can be used to define the interval modalities $\Box$ (everywhere) and $\Diamond$ (somewhere). Atomic formulas in DC include equalities and arithmetic inequalities over DC terms. Terms in DC are constructed using arithmetic operations such as addition, subtraction, multiplication, and division. Terms are also evaluated over intervals. The main novelty in DC is the duration operator $\int s$, where s is a *state* formula, i.e., a predicate over the time parameter whose truth value varies only finitely often in any bounded time interval. On the interval $[b, e]$, $\int s$ represents the Riemann integral $\int_b^e s(t).dt$, that is it returns the accumulated time duration over the interval $[b, e]$ for which s has been true. In particular $\int \mathbf{1}$ is just the length of the interval $e - b$ and is represented by the term l.

DC can, for example, be used to prove a safety property of a gas burner that if gas is allowed to leak for no more than 1 second in a 30 second interval, then for any duration longer than 60 seconds, gas is leaking at most 1/20th of the time.

Skakkebæk [37,38], as part of his doctoral work, defined the semantics of DC within PVS, used the semantics to validate the inference rules of the Duration Calculus, and developed PVS strategies for applying the DC proof rules.[6] The resulting system PC/DC delivers the benefits of the arithmetic, equality, and

[6] The duration operator was actually defined axiomatically but the rest of the interval temporal logic was presented semantically.

typechecking capabilities of PVS within the language and notational framework of DC. Inal and Skakkebæk [23] have applied PC/DC to a several previously studied DC examples. They report finding errors in every example that they examined.

In the example of the gas burner, the requirements and design can be stated and typechecked in the DC dialect of PC/DC as shown below.

```
dcgasburner: DCTHEORY
BEGIN
  IMPORTING dcp_thms
  Leak: State

  Safe: IForm = n(20)*dur(Leak) <= len
  Req: IForm = len >= n(60) => Safe

  Des_1: IForm = [](`Leak` => len <= n(1))
  Des_2: IForm = []((<>`Leak`);(<>`not Leak`);(<>`Leak`) => len >= n(30))

  T: IForm = n(30)*dur(Leak) <= len
  Triple: IForm = T \/ (T;`Leak`) \/ (T;`Leak`;`not Leak`)

  lem1: LEMMA Des_1 /\ Des_2 => Triple
  lem2: LEMMA Des_1 /\ Triple /\ (len >= n(60)) => Safe

  Gasburner: LEMMA (Des_1 /\ Des_2 /\ (len >= n(60))) => Safe
END dcgasburner
```

PC/DC proofs allow all the commands of PVS in addition to the DC inference rules. Subgoals in a PC/DC proof are also maintained in the DC dialect as shown in the example sequent below.

```
Preserve :

{-1}   (30 * dur(Leak) <= dur(tt));(30 * dur(Leak) <= dur(tt))
  |-------
{1}   30 * dur(Leak) <= dur(tt)
```

In addition to reinforcing the claims made in Sections 2 and 3, the above embedding shows how the deduction and typechecking capabilities can be exploited to support specification and verification in a special-purpose program logic. In this way, a useful general-purpose DC verification system could be constructed without the exorbitant overhead of building all the language and deduction components from scratch.

5 Tabular Specifications

We have already seen how special-purpose program logics and their associated deductive procedures can be integrated into PVS. This section presents the enhancement of PVS with support for tabular specification [31]. This enhancement

yields a visually attractive specification notation that is supported by the specification and deductive capabilities already in PVS. In particular, PVS typechecking is used to generate well-formedness proof obligations. These proof obligations can be verified using the PVS proof checker. Furthermore, the model checking capabilities of PVS described in Section 3 can be used to verify and refine tabular specifications of mode transition systems.

A number of influential specification and verification methodologies are based on the use of tables [10, 17, 18, 26, 36]. There are a few special purpose tools that support the analysis of tabular specifications like SCR* [15, 16], RSML [13, 14], and Tablewise [20]. Since these tools are not attached to a general-purpose verification system, the analysis capabilities of these tools are quite limited.

A simple tabular specification can be given in terms of condition/entry pairs, as in the following specification of the sign of a number.

$$sign(x) = \begin{array}{|c|c|c|} \hline x < 0 & x = 0 & x > 0 \\ \hline -1 & 0 & +1 \\ \hline \end{array}$$

This table can be directly captured by means of the newly added PVS construct `COND` which captures conditional branching.

```
signs: TYPE = { x: int | x >= -1 & x <= 1}

x: VAR int

sign_cond(x): signs =
  COND
    x < 0  -> -1,
    x = 0  ->  0,
    x > 0  ->  1
  ENDCOND
```

Typechecking the above specification generates proof obligations requiring that the conditions of a `COND` construct be pairwise-disjoint and exhaustive. Internally, the PVS theorem prover treats the `COND` as a nested `IF-THEN-ELSE` so that the only special treatment for `COND` expressions is in the PVS typechecker. A visually more attractive specification of the `COND` expression above can be given using the newly added `TABLE` construct where the start of a condition, `->`, and `,` are replaced by `|`, `|`, and `||`, respectively. Note that the text following `%` in a line are comments that are placed to enhance readability.

```
sign_vtable(x): signs = TABLE
                %-------------%
                | x < 0 | -1 ||
                %-------------%
                | x = 0 |  0 ||
                %-------------%
                | x > 0 |  1 ||
                %-------------%
ENDTABLE
```

There are a few syntactic variants of the TABLE construct. When the conditions are enumerative, i.e., they are equalities of the form $z = a_i$ over subrange or enumerated type elements a_i, then the z parameter can be extracted out as a column heading.

```
modes: TYPE = { off, armed, engaged }

value(m:modes):bool = TABLE
          m
        %------------------%
        | off      | false ||
        %------------------%
        | armed    | true  ||
        %------------------%
        | engaged  | true  ||
        %------------------%
ENDTABLE
```

Other variants of the TABLE construct allow the specification of horizontal 1-dimensional tables where conditions appear as column headings, 2-dimensional conditional or enumerative tables. Blank entries in tables are treated as unreachable and the corresponding proof obligations are generated by the PVS type checker.

The TABLE construct has been used to capture an SCR-style specification [2] of the mode-transition table of a simple cruise-controller where the modes (`off`, `inactive`, `cruise`, `override`) are controlled by the environment conditions (`ignited`, `running`, `toofast`, `brake`, `activate`, `deactivate`, `resume`). The table indicates the next mode corresponding to the current mode and the status of the environment conditions. The tabular specification generates well-formedness proof obligations that can be verified using the PVS proof checker. Deeper assurance regarding the correctness of the specification can be obtained by applying CTL model checking to check such properties as:

1. When the mode is `off`, the ignition is off (i.e., `ignited` is *false*).
2. In modes other than `off`, the ignition is on (i.e., `ignited` is *true*).
3. In `inactive` mode, either the engine is not `running` or the cruise control is not `activated`.
4. In `cruise` mode, the engine is `running`, the vehicle is not going `toofast`, the `brake` is not on, and `deactivate` is not selected.
5. In `override` mode, the engine is `running`.

Atlee and Gannon [2] used a hand-translation of the SCR specification into SMV [29] in order to perform model checking, whereas the tabular specification in PVS is directly combined with the PVS model checker.

Tabular specification and verification have also been applied to a toy autopilot specification where there are two interacting components: one component

sets the climb angle, and the other sets the target altitude. We can use model checking to prove safety and liveness properties of a tabular specification of the auto-pilot system and also to demonstrate that one tabular specification implements another. PVS tables were used to capture the quotient lookup table in the verification of hardware for SRT division [35]. These example applications illustrate the value of integrating tabular notation into a general-purpose verification framework. Through this integration, we can exploit the visual appeal and simplicity of tables in system specification, and also employ sophisticated deductive tools in the analysis of tabular specifications.

6 Conclusions

I have argued that while special-purpose formalisms and verification tools are important, an attempt should be made to integrate them within a general-purpose framework based on conventional mathematics. I have given illustrative examples where symbolic model checking has been integrated into PVS as a decision procedure, the Duration Calculus has been embedded as a notational front-end to PVS, and PVS has been enhanced with support for tabular specification. In each case, the combination exhibited a useful synergy between generality and specificity.

What does all this have to do with fault-tolerant and real-time systems? PVS has been used in the verification of several fault-tolerant and real-time systems. These examples rely quite heavily on the conventional mathematical capabilities of PVS (e.g., equational and arithmetic decision procedures, induction, rewriting). The amount of mathematics can be quite overwhelming if it is not organized into theories, libraries, and special-purpose formalisms. A number of other users have similarly employed PVS to support formalisms that they previously employed by hand [1, 19, 25].

PVS, like other similar systems, was not explicitly designed with the anticipation of issues of integration and unification. It was in fact designed as a framework for developing conventional mathematics that integrated the use of decision procedures, rewriting, induction, and simplification with typechecking. The integrated nature of PVS has proved useful in the definition and integration of more specialized logics and deductive tools, but such integration work still has to be carried out in a somewhat *ad hoc* manner.

Logical frameworks like Isabelle [33] offer considerable convenience in defining and constructing deductive tools for specialized logics, but lack any systematic support for a coherent integration of the capabilities of different formalisms. The clear technical challenge is to design a system that offers the definitional convenience of logical frameworks together with coherent integration and interaction between different formalisms and deductive tools as demonstrated by the examples shown in this paper.

In the end, the basic difficulty of formal methods lies in the sheer scale of mathematics that is needed. Models, methods, tools, and notations provide

useful navigational aids but these are not all that useful in isolation. Unification of the sort described in this paper is a necessary step towards overcoming the problem of scale.

References

1. M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *IEEE Real-Time Technology and Applications Symp. (RTAS'96)*, Boston MA, June 1996. IEEE Computer Society Press. To Appear.
2. Joanne M. Atlee and John Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.
3. J. R. Burch, E. M. Clarke, K. L McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *5th Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, PA, June 1990. IEEE Computer Society.
4. Zhou Chaochen, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1992.
5. E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. In David Dill, editor, *Computer-Aided Verification 94*, volume 818 of *Lecture Notes in Computer Science*, pages 415–427, Stanford, CA, June 1994. Springer Verlag.
6. E. M. Clarke and O. Grumberg. Research on automatic verication of finite state concurrent systems. In *Annual Review of Computer Science*, pages 269–290. Annual Reviews, Inc., 1987.
7. David Cyrluk, Patrick Lincoln, Steven Miller, Paliath Narendran, Sam Owre, Sreeranga Rajan, John Rushby, Natarajan Shankar, Jens Ulrik Skakkebæk, Mandayam Srivas, and Friedrich von Henke. Seven papers on mechanized formal verification. Technical Report SRI-CSL-95-3, Computer Science Laboratory, SRI International, Menlo Park, CA, January 1995.
8. Bruno Dutertre. Elements of mathematical analysis in PVS. In *International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, Turku, Finland, August 1996. Springer-Verlag. To appear.
9. E.A. Emerson and C.L Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the 10th Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, LA, January 1985. Association for Computing Machinery.
10. S. Faulk and P. Clements. The NRL Software Cost Reduction (SCR) requirements specification methodology. In *Fourth International Workshop on Software Specification and Design*, Monterey, CA, April 1987. IEEE Computer Society.
11. Michael J. C. Gordon. Mechanizing programming logics in higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Theorem Proving*, pages 387–439, New York, NY, 1989. Springer-Verlag.
12. Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, number 1051 in Lecture Notes in Computer Science, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.

13. Mats P. E. Heimdahl. Experiences and lessons from the analysis of TCAS II. In Steven J. Zeil, editor, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 79–83, San Diego, CA, January 1996. Association for Computing Machinery.
14. Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.
15. Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In COMPASS [22], pages 109–122.
16. Constance Heitmeyer, Bruce Labaw, and Daniel Kiskis. Consistency checking of SCR-style requirements specifications. In *International Symposium on Requirements Engineering*, York, England, March 1995. IEEE Computer Society.
17. K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.
18. K. L. Heninger et al. Software requirements for the A-7E aircraft. NRL Report 3876, Naval Research Laboratory, November 1978.
19. Jozef Hooman. Correctness of real time systems by construction. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 19–40, Lübeck, Germany, September 1994. Springer-Verlag.
20. D. N. Hoover and Zewei Chen. Tablewise, a decision table tool. In COMPASS [22], pages 97–108.
21. John Hopcroft and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979.
22. *COMPASS '95 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1995. IEEE Washington Section.
23. Recep Inal and Jens U. Skakkebæk. Applying a mechanized duration calculus assistant. In Hans Rischel, editor, *Nordic Seminar on Dependable Computing Systems*, pages 69–80, Lyngby, Denmark, August 1994. Technical University of Denmark.
24. G. Janssen. *ROBDD Software.* Department of Electrical Engineering, Eindhoven University of Technology, October 1993.
25. Pertti Kellomäki. Mechanical verification of DisCo specifications. In *Israeli-Finnish Binational Symposium on Specification, Development, and Verification of Concurrent Systems*, Haifa, Israel, January 1996. The Technion.
26. Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.
27. Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer-Verlag.
28. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification.* Springer-Verlag, New York, NY, 1992.
29. Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Boston, MA, 1993.
30. S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

31. Sam Owre, John Rushby, and Natarajan Shankar. Analyzing tabular and state-transition specifications in PVS. Technical Report SRI-CSL-95-12, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1995. Available, with specification files, from `http://www.csl.sri.com/csl-95-12.html` .
32. David Park. Finiteness is mu-ineffable. *Theoretical Computer Science*, 3:173–181, 19.
33. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
34. S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In Pierre Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.
35. H. Rueß, N. Shankar, and M.K. Srivas. Modular verification of SRT division. In R. Alur and T.A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, Lecture Notes in Computer Science, New Brunswick, NJ, July 1996. Springer-Verlag. To appear.
36. Lance Sherry. A structured approach to requirements specification for software-based systems using operational procedures. In *13th AIAA/IEEE Digital Avionics Systems Conference*, pages 64–69, Phoenix, AZ, October 1994.
37. Jens U. Skakkebæk and N. Shankar. A Duration Calculus proof checker: Using PVS as a semantic framework. Technical Report SRI-CSL-93-10, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.
38. Jens Ulrik Skakkebæk. *A Verification Assistant for a Real-Time Logic*. PhD thesis, Department of Computer Science, Technical University of Denmark, Lyngby, Denmark, November 1994.
39. Mandayam K. Srivas and Steven P. Miller. Formal verification of the AAMP5 microprocessor. In Michael G. Hinchey and Jonathan P. Bowen, editors, *Applications of Formal Methods*, Prentice Hall International Series in Computer Science, chapter 7, pages 125–180. Prentice Hall, Hemel Hempstead, UK, 1995.
40. Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA, 1990.

Abstract and Model Check while you Prove*

To be presented at the eleventh International Conference on Computer-Aided Verification (CAV99), Trento, Italy, Jul 7-10, 1999

Hassen Saïdi and Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA
{saidi,shankar}@csl.sri.com

Abstract. The construction of abstractions is essential for reducing large or infinite state systems to small or finite state systems. Boolean abstractions, where boolean variables replace concrete predicates, are an important class that subsume several abstraction schemes. We show how boolean abstractions can be constructed simply, efficiently, and precisely for infinite state systems while preserving properties in the full μ-calculus. We also propose an automatic refinement algorithm which refines the abstraction until the property is verified or a counterexample is found. Our algorithm is implemented as a proof rule in the PVS verification system. With the abstraction proof rule, proof strategies combining deductive proof construction, model checking, and abstraction can be defined entirely within the PVS framework.

1 Introduction

When verifying temporal properties of reactive systems, algorithmic methods are used when the problem is decidable, and deductive methods are employed, otherwise. Algorithmic methods such as model checking are limited by the state space explosion problem. State space reduction techniques such as symbolic representations, symmetry, and partial order reductions have yielded good results but the state spaces that can be handled in this manner are still quite modest. Deductive methods using theorem proving continue to require a considerable amount of manual guidance. While it is clear that any way out of this impasse must rely on a combination of theorem proving and model checking, specific methodologies are needed to make such a combination work with a reasonable degree of automation. It is known that abstraction is a key methodology in combining deductive and algorithmic techniques. Abstraction can be used to reduce problems to model-checkable form, where deductive tools are used to construct

* This research was supported by the National Science Foundation under Grant Nos. CCR-9509931 and CCR-9712383, and by the Air Force Office of Scientific Research Contract No. F49620-95-C0044. We thank our colleagues John Rushby and Sam Owre for their helpful comments on earlier versions of this paper.

valid abstract descriptions or to justify that a given abstraction is valid. In this paper, we propose a practical verification methodology that is, based on a simple, efficient, and precise form of boolean abstraction generation that preserves properties in the μ-calculus. We extend the boolean abstraction scheme defined in [GS97] that uses predicates over concrete variables as abstract variables, to abstract assertions in the rich assertional language of PVS [OSRSC98]. The PVS language admits the definition of a fixed point operator that is used to define the μ-calculus in PVS [RSS95]. With this definition of the μ-calculus in PVS, model checking implemented as a PVS proof rule can be used as a decision procedure.

Our conservative abstraction scheme is implemented as a proof rule that abstracts any PVS formula over concrete state variables and produces a PVS formula over abstract state variables. Any assertion expressing a general or temporal property of a concrete PVS specification is abstracted into a *stronger* assertion expressing a property over the corresponding abstract specification. The resulting abstract assertion is in a decidable logic, and decision procedures such as model checking can be used to discharge it.

Unlike previous work for the automatic abstraction of infinite state systems using decision procedures [GS97,CU98,BLO98], our algorithm does not always over-approximate the transition relation as is done to preserve only universally quantified path temporal formulas in logics such as $\forall$CTL. Extensions of the preservation results [DGG94,CGL94] to the more expressive logic CTL* are defined using the notion of mixed abstraction which involves multiple next-state relations. Our algorithm abstracts a μ-calculus formula which is not tied to a single transition system. Thus, no distinction is made between universal and existential fragments. The integration of our abstraction algorithm as a PVS proof rule allows us to design powerful proof strategies combining abstract interpretation, model checking and proof checking. We also propose an automatic abstraction refinement algorithm that is applied when model checking fails. This is done by automatically enriching the abstract state with new relevant predicates until the property is proved or a counterexample is found.

The paper is organized as follows. In Section 2 we show how boolean abstractions can be defined in PVS. In Section 3, we present an efficient abstraction algorithm for the computation of the "most precise" abstraction of a given boolean abstraction of a predicate over concrete state variables. In Section 4, we generalize this algorithm to abstract any PVS assertion, including μ-calculus formulas over concrete state variables into assertions over abstract state variables. In Section 5, we present the refinement algorithm.

2 Boolean Abstractions in PVS

Propositional μ-calculus is an extension of propositional calculus that includes predicates defined by means of least and greatest fixed point operators, μ and ν, respectively. It is strictly more expressive than CTL* which includes both linear and branching time temporal logics such as LTL and CTL. In [RSS95] a detailed

description of the encoding of the propositional μ-calculus in PVS is presented. The least fixed point operator is defined as $\mu(F) = \bigcap\{x \mid F(x) \subseteq x\}$, the predicate that is the greatest lower bound of the pre-fixed points of a monotone predicate transformer F. The temporal operators of CTL, such as **AG**, **AF**, **EG**, and **EF**, can be easily defined using their fixed-point characterizations. When the state space is finite, the predicates can be coded in boolean form and model checking of μ-calculus formulas can be done using binary decision diagrams (BDDs).

As a simple example, we consider a simple protocol where two processes are competing to enter a critical section in mutual exclusion using a semaphore. The PVS theory describing the protocol is given as follows.

```
semaphore : THEORY
  BEGIN
IMPORTING  MU@ctlops
 location : TYPE = {idle, wait, critical}
 state : TYPE = [# pc1,pc2:location , sem: int #]
  s,s1,s2 : VAR state

 init(s) : bool= pc1(s)=idle and pc2(s)=idle and sem(s)=1

 N(s1,s2) : bool = ...

  safe: THEOREM
   init(s) IMPLIES
    AG(N, LAMBDA s: NOT (critical?(pc1(s)) AND
                         critical?(pc2(s))))(s)
END semaphore
```

The state is given as a record consisting of two program counters and a semaphore `sem`. The expression `N(s1,s2)` is transition relation of the protocol. We are interested in proving that both processes have mutually exclusive access to the critical section. The property `safe` is expressed as a CTL property using the usual operator **AG**, which is translated into a μ-calculus property. When the state type is finite, the property can be verified using model checking[RSS95]. In this simple example, `sem` is of type integer and cannot be encoded with a finite number of boolean variables and hence the property cannot be directly model checked. We propose to extend the capabilities of PVS with a boolean abstraction mechanism that can conservatively reduce a μ-calculus property of an infinite state system to model checkable form. In this abstraction, certain predicates at the concrete level (that might be used in guards, expressions, or properties) can be replaced by abstract boolean variables. This gives us a general method for constructing abstractions by evaluating any predicate over the variables of the program. Since the set of boolean variables is finite, so is the set of abstract states. Boolean abstraction is defined using a set of predicates of the form $\lambda(s : \texttt{state}) : \varphi(s)$ over the `concrete` state type `state`. An abstraction of the mutual exclusion protocol can be defined using two predicates

$\lambda(s) : sem(s) \leq 0$ and $\lambda(s) : sem(s) > 0$. These predicates define an abstract state type

$$\text{abs\_state} : \text{TYPE} = [\# \text{ pc1}, \text{pc2} : \text{location}, \text{B1}, \text{B2} : \text{boolean}\#]$$

where the state components `pc1` and `pc2` are of finite type and therefore are not abstracted, and the state component `sem` referenced by the two predicates defining the abstraction is encoded with two boolean components `B1` and `B2` corresponding to the two predicates. In this particular example, these two predicates happen to be exclusive, but boolean abstractions can be defined more generally with an arbitrary set of predicates over the concrete state type.

3 Efficient Computation of Boolean Abstractions

Abstract interpretation [CC77] is the general framework for defining abstractions using Galois connections[1]. The domain of the abstraction function α consists of sets of concrete states, represented by predicates, and ordered by implication. The range of the abstraction consists of boolean formulas constructed using the boolean variables $B_1, \cdots, B_k$, ordered by implication. If X ranges over sets of concrete states and Y ranges over boolean formulas in $B_1, \cdots, B_k$, then the abstraction and concretization function α and γ have the following properties:

- $\alpha(X) = \bigwedge\{Y \mid X \Rightarrow \gamma(Y)\}$,
- $\gamma(Y) = \bigvee\{X \mid \alpha(X) \Rightarrow Y\}$.

However, we use a simpler and precise concretization function γ which consists simply in substituting each abstract variable B_i by its corresponding predicate φ_i, and each abstract state variable `abs_s` by the corresponding concrete state variable `s`. That is

$$\gamma(Y) = Y[\varphi_i(s)/B_i(abs\_s)].$$

We propose to apply boolean abstractions to any predicate (assertion or transition relation) written in a rich assertional language.

Abstraction of assertions. For any predicate P over the concrete variables, the abstraction $\alpha(P)$ of P can be computed as the conjunction of all boolean expressions b satisfying the condition:

$$P \Rightarrow \gamma(b) \tag{1}$$

[1] A Galois connection is a pair (α, γ) defining a mapping between a concrete domain lattice $\wp(\mathcal{Q})$ and an abstract domain lattice $\wp(\mathcal{Q}^A)$, where α and γ are two monotonic functions such that $\forall(P_1, P_2) \in \wp(\mathcal{Q}) \times \wp(\mathcal{Q}^A)$. $\alpha(P_1) \subseteq P_2 \Leftrightarrow P_1 \subseteq \gamma(P_2)$.

Note that there are 2^{2^k} distinct boolean truth functions in k variables, and testing all of these could become very expensive. This set is designated as the set of *test points*. An abstraction is *precise* with respect to the considered abstract lattice, if the set of test points is the entire set of the boolean expressions forming the abstract lattice. Any over-approximation of the $\alpha(P)$ can be computed with a smaller set of test points for which the implication (1) must be valid. For example, in [GS97], the abstract lattice considered is the lattice of monomials[2] over the set of boolean variables. In this case, it is not necessary to prove (1) for all the monomials over the set $\{B_1, \cdots, B_k\}$, but only for the atoms $B_1, \cdots, B_k$ and their negations. We can efficiently compute $\alpha(P)$ for any predicate P by choosing the abstract space as the whole boolean algebra over B or by choosing a sub-lattice of B and the corresponding test points, using the following fact:

Theorem 1. *Let $B = \{B_1, \cdots, B_k\}$ be a set of boolean variables, and let $\mathcal{B}_A$ be the boolean algebra defined by the structure $< B, \wedge, \vee, \neg, true, false >$. Let $\mathcal{D}_B$ be the subset of $\mathcal{B}_A$ containing only literals[3] and disjunctions of literals. To compute the most precise image by α of any set of concrete states P (given as a predicate), it is sufficient to consider as a set of test points, the set $\mathcal{D}_B$ instead of the whole set $\mathcal{B}_A$ of boolean expressions. That is, testing*

$$P \Rightarrow \gamma(b)$$

for all boolean expressions in $\mathcal{B}_A$ is equivalent to test this implication only for b in $\mathcal{D}_B$. That is, 2^{2^k} tests can be reduced to at most only $3^k - 1$ tests.

Proof. We consider the fact that each boolean expression b can be written in a conjunctive normal form $d_1 \wedge \cdots \wedge d_j$, where each d_i is a disjunction of literals. Thus, the proof of the implication (1) for each element b can be first decomposed to simpler proofs $P \Rightarrow \gamma(d_i)$. This implication can be proved for each d_i by first testing one disjunct, that is a literal, or more than one disjunct if necessary. That is, only for disjunctions in $\mathcal{D}_B$. ∎

This theorem gives us an efficient way of computing precise abstractions by reducing the set of proof obligations from 2^{2^K}, the number of elements of $\mathcal{B}_A$, to only $3^k - 1$, the number of elements of the smaller set $\mathcal{D}_B$, and also gives us an order in which the proof obligations should be generated and proved. In fact, when the set of predicates $\{\varphi_1, \cdots, \varphi_k\}$ is properly chosen, the actual number of tests is far fewer than $3^k - 1$. When a proof for any element b_i of the set $\mathcal{D}_B$ succeeds or fails, then the number of tests will decrease due to the fact that for many elements b_j of $\mathcal{D}_B$, the test is redundant due to subsumption. Figure 1(a) shows how the image by α of a set $P(s)$ of concrete states is computed. The variable α is initialized to *true*. The variable *fail* consists of the set of elements of $\mathcal{D}_B$ that have not been proved to be in the abstraction of P. The set *fail* is

---

[2] Monomial are the expressions $\bigwedge_{i \in \{1 \cdots k\}} b_i$ where each b_i is either B_i or $\neg B_i$.

[3] A literal is either a boolean variable B_i or its negation $\neg B_i$

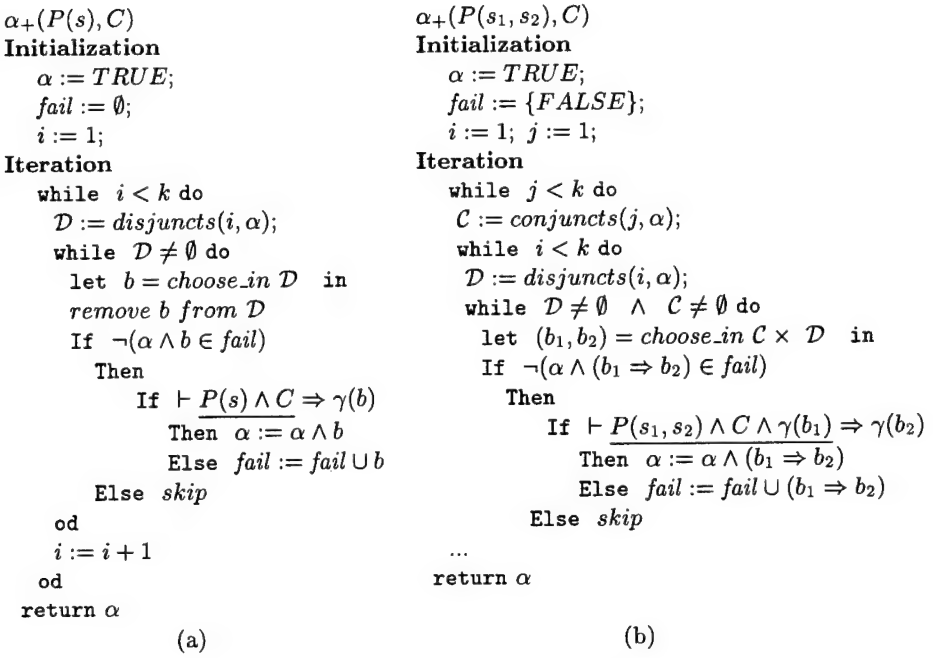


Fig. 1. Efficient computation of $\alpha(P)$

initially just the singleton $\{FALSE\}$. It is assumed that there has already been a prior check to ensure that $P(s) \wedge C$ is not equivalent to $FALSE$. The construction starts by using disjunctions of length 1, i.e., the literals B_i and $\neg B_i$ for b. The literals b for which the proof obligation $P(s) \Rightarrow \gamma(b)$ succeeds, are added to α. At each iteration, when such a proof succeeds, it is possible to eliminate from the current set of test points the elements for which the test is no longer necessary. This is done by the test $\alpha \wedge b \in fail$. For instance, in the first iteration when we consider only literals, if the proof succeeds for B_i, it is not necessary to test $\neg B_i$. The test for $\neg B_i$ can only fail, otherwise, both $\neg B_i$ and B_i would be added to α, and $\alpha(P)$ would be equivalent to $FALSE$. In the next iteration, the test points that are disjunctions of two literals and not already subsumed by the disjunctions in α, are considered. Once again, the successful test points are added to α, i is incremented and the iteration is repeated for disjunctions of length i. The image α of a set of concrete states is computed incrementally and can be interrupted at any moment, providing an over-approximation of the precise image. Furthermore, we use additional heuristics to avoid unnecessary tests. For instance, if the intersection of the set of free variables of P and those of $\gamma(B_i)$ is empty, it is not necessary to consider the boolean expressions constructed using B_i.

Abstraction of a Transition Relation. Transitions are expressed as general assertions over a pair of concrete states (s_1, s_2). The abstraction of a predicate $P(s_1, s_2)$ describing such a transition relation is defined as a predicate $B(abs\_s_1, abs\_s_2)$ over the abstract pair (abs_s_1, abs_s_2). Figure 1(b) shows how a concrete predicate $P(s_1, s_2)$ representing a transition relation is abstracted. The algorithm constructs a transition relation over the variables $\{B_1, \cdots, B_{2k}\}$ by constraining the current and the next abstract states. This is done by considering as set of test points the set of implications $b_1 \Rightarrow b_2$, where b_1 and b_2 represent formulas in the current and the next abstract state variables, respectively. Again, the abstraction of P is computed incrementally by first constraining the next state, that is by enumerating the disjunctions b_2. When all the proofs fail for a given choice of b_1, the current state is constrained by considering a longer conjunction for b_1. Consider for instance the expression

$$s_2 = s_1 \text{WITH } [\text{sem} := \text{sem}(s_1) + 1].$$

This assertion over a pair of concrete state variables (s_2, s_2) of type `state` is abstracted with respect to the predicates $\lambda(s) : sem(s) \leq 0$ and $\lambda(s) : sem(s) > 0$ to the following assertion over a pair of abstract state variables (abs_s_1, abs_s_2) of type `abs_state`:

$$\begin{array}{l}
\quad (B_1(abs_s_1) \Rightarrow (B_1(abs_s_2) \vee B_2(abs_s_2))) \\
\wedge\ (B_2(abs_s_1) \Rightarrow B_2(abs_s_2)) \\
\wedge\ (\neg B_2(abs_s_1) \Rightarrow (\neg B_2(abs_s_2) \vee \neg B_1(abs_s_2)) \wedge (B_1(abs_s_2) \vee B_2(abs_s_2))) \\
\wedge\ (\neg B_1(abs_s_1) \Rightarrow B_2(abs_s_2) \wedge \neg B_1(abs_s_2)).
\end{array}$$

4 Abstract Interpretation as a Proof Rule

Our abstraction algorithm computes the most precise over-approximation of an assertion over concrete states, using a validity checker for the generated assertions. We implemented this algorithm in the PVS verification system as a primitive proof rule. Our goal is to approximate a PVS formula over concrete state variables, that is a PVS boolean expression, by a formula over abstract state variables. This generated theorem is stronger than the original one. However, it is expressed in a decidable theory that can be handled by model-checking, BDD simplification, or the ground decision procedures available in PVS. To do so, we generalize the abstraction algorithm defined in [PH97] for the μ-calculus to the PVS assertion language and we use our abstraction algorithm to approximate assertions. This algorithm abstracts propositional μ-calculus formulas using over-approximation of predicates and under-approximation of negated predicates. Under-approximation of an assertion is defined as follows:

$$\alpha\_(P(s)) = \bigvee \{b \mid \gamma(b) \Rightarrow P(s)\}$$

We use only the over-approximation algorithm relying on the following lemma.

Lemma 1. *Let φ a predicate defining a set of states. For all predicate φ*

$$\alpha_+(\neg\varphi(s)) \Leftrightarrow \neg\alpha_-(\varphi(s)).$$

We now formally define the abstraction function $[\![\ ]\!]^\sigma$ which approximates a PVS boolean expression f such that, $[\![\ f\ ]\!]^+$ denotes an *over* approximation of f, and $[\![\ f\ ]\!]^-$ an *under* approximation of f. We also use a context c consisting of a PVS formula that is valid at the PVS subformula that is being approximated. The intuition behind using such a context expression is that when an expression $e_1 \wedge e_2$ is being abstracted, one can assume that e_1 is valid when abstracting e_2 and vice-versa. The context when omitted is just the boolean constant *TRUE*. $[\![\ f\ ]\!]^\sigma_c$ denotes the approximation of f under the context c.

Approximation of PVS assertions. The abstraction function $[\![\ ]\!]$ is defined recursively on the structure of the PVS assertion language as follows.

$$\begin{array}{lll}
\textit{propositions}: & [\![e_1 \wedge e_2]\!]^\sigma_c & \longrightarrow [\![e_1]\!]^\sigma_{c\wedge e_2} \wedge [\![e_2]\!]^\sigma_{c\wedge e_1} \\
& [\![\neg e]\!]^\sigma_c & \longrightarrow \neg[\![e]\!]^{-\sigma}_c \\
\\
\textit{quantifiers}: & [\![\exists(s) : e]\!]^\sigma_c & \longrightarrow \exists(abs_s) : [\![e]\!]^\sigma_c \\
& [\![\forall(s) : e]\!]^\sigma_c & \longrightarrow \forall(abs_s) : [\![e]\!]^\sigma_c \\
& [\![\lambda(s) : e]\!]^\sigma_c & \longrightarrow \lambda(abs_s) : [\![e]\!]^\sigma_c \\
\\
\textit{fixpoints}: & [\![\mu/\nu(\lambda(Q) : \mathtt{F}(Q))]\!]^\sigma_c & \longrightarrow \mu/\nu(\lambda(abs_Q) : [\![\mathtt{F}(Q)]\!]^\sigma_c) \\
\\
\textit{atoms}: & [\![e(s)]\!]^+_c & \longrightarrow \alpha_+(e(s), c) \\
& [\![e(s_1, s_2)]\!]^+_c & \longrightarrow \alpha_+(e(s_1, s_2), c) \\
& [\![e(s)]\!]^-_c & \longrightarrow \alpha_-(e(s), c) \\
& [\![e(s_1, s_2)]\!]^-_c & \longrightarrow \alpha_-(e(s_1, s_2), c) \\
& [\![\varphi_i(s)]\!]^\sigma_c & \longrightarrow B_i(abs_s) \\
\\
\textit{constants}: & [\![\ e\ ]\!]^\sigma_c & \longrightarrow e \quad \text{if } \textit{free variables}(e) = \emptyset
\end{array}$$

The following theorem establishes the fact that the abstraction provides, respectively, an over and under approximation of any PVS boolean expression.

Theorem 2. *Let f be a PVS assertion, $[\![\ ]\!]$ an abstraction function. We have:*

$$\vdash f \Rightarrow \gamma([\![\ f\ ]\!]^+) \quad \textit{and} \quad \vdash \gamma([\![\ f\ ]\!]^-) \Rightarrow f$$

Proof. The proof is established by induction on the structure of the assertion f. It is easy to show that by the definitions of α_+ and α_-, both implications hold when f is an atom. The other cases can be deduced by monotonicity of the logical connectives, and the fixed point operators. ∎

The soundness of the abstraction function is established by the following theorem.

Theorem 3 (preservation). *Let* $[\![\ ]\!]$ *be the abstraction function defined above as a boolean abstraction, and let* f *be any PVS boolean formula. Then*

$$\vdash [\![f]\!]^- \quad \textit{implies} \quad \vdash f$$

Theorem 2 ensures that for an assertion f, the abstraction algorithm produces a stronger assertion $\gamma([\![f]\!]^-)$. Note that $\vdash [\![f]\!]^-$ trivially implies $\vdash \gamma([\![f]\!]^-)$, which then justifies the preservation result of Theorem 3.

The abstraction algorithm where a formula f is under-approximated is implemented as a PVS proof rule `abstract`. This atomic proof rule takes a goal given by a PVS formula (a μ-calculus formula) and a set of state predicates, and translates this to a propositional formula (a propositional μ-calculus formula) which is returned as a new goal. This goal can be discharged using any other PVS proof command including BDD simplification and model checking.

We have defined a PVS proof strategy that carries out a sequence of inference steps that simplify goal formulas by rewriting all definitions, including constant definitions such as the temporal operators of the logic CTL in terms of the μ and ν operators, and applies the abstraction function on the resulting goal.

$$\begin{array}{l}
\forall\ (s:\ \text{state}):\\
\ \text{init}(s) \supset\\
\ \ \neg\mu.\lambda\ (Q:\ \text{pred[state]}):\\
\ \ \ \ (\lambda\ (u:\ \text{state}):\\
\ \ \ \ \ (\neg\lambda\ s:\\
\ \ \ \ \ \ \ \neg(\text{critical?}(\text{pc1}(s))\wedge\\
\ \ \ \ \ \ \ \ \ \text{critical?}(\text{pc2}(s))))\\
\ \ \ \ \ (u)\vee\\
\ \ \ \ \ \exists\ (v:\ \text{state}):\\
\ \ \ \ \ \ (Q(v)\wedge N(u,v)))(s)
\end{array}
\qquad
\begin{array}{l}
\forall\ (abs_s:\ \text{abs_state}):\\
\ \neg[\![\text{init}(s)]\!]^+\vee\\
\ \ \neg\mu.\lambda\ (abs_Q:\ \text{pred[abs_state]}):\\
\ \ \ \ (\lambda\ (abs_u:\ \text{abs_state}):\\
\ \ \ \ \ (\neg\lambda\ abs_s:\\
\ \ \ \ \ \ \ \neg(\text{critical?}(\text{pc1}(abs_s))\wedge\\
\ \ \ \ \ \ \ \ \ \text{critical?}(\text{pc2}(abs_s))))\\
\ \ \ \ \ (abs_u)\vee\\
\ \ \ \ \ \exists\ (abs_v:\ \text{abs_state}):\\
\ \ \ \ \ \ (abs_Q(abs_v)\wedge[\![N(u,v)]\!]^+))(abs_s)
\end{array}$$

Fig. 2. An example of abstraction for a PVS assertion

Figure 2 shows how the μ-calculus formula corresponding to the theorem `safe` presented in the PVS theory `semaphore` in Section 2 is approximated. The property of mutual exclusion $\lambda(s)\ :\ \neg(\text{critical?}(\text{pc1}(s))\wedge\text{critical?}(\text{pc2}(s)))$ is expressed as an invariance property. As expected for such properties, the initial state and the transition relation are over-approximated. For instance, we have

$$[\![\text{init}(s)]\!]^+ \longrightarrow \text{idle?}(\text{pc1}(abs\_s))\wedge\text{idle?}(\text{pc2}(abs\_s))\wedge\neg B_1(abs\_s)\wedge B_2(abs\_s)$$

We have tried other examples including a simple snoopy cache-coherence protocol with an arbitrary number of processes [Rus97] and a variant of the alternating-bit communication protocol called the bounded retransmission protocol [HS96]. The main invariant of the cache coherence protocol is proved by an

abstraction defined in terms of five predicates. The preservation of the invariant is then proved by abstraction and BDD-based propositional simplification.

The bounded retransmission protocol is verified using an abstraction also defined in terms of five predicates. The construction of the abstract description takes about 100 seconds in PVS. The resulting abstract assertion is discharged using model checking. In contrast, Havelund and Shankar's verification [HS96] of this example required 57 invariants to justify the validity of a manually derived abstraction.

5 Refining an Abstraction

The abstraction proof rule is used in PVS to generate new goals that depend only on finite state variables. Such goals can be discharged using a PVS proof rule such as the BDD simplifier or the μ-calculus simplifier. However model checking on the new goal can fail because the abstraction is too coarse. It is then necessary to refine the abstraction using a richer abstract domain. Since our abstraction algorithm presented in Section 4 allows us to compute the most precise abstraction with respect the predicates $\varphi_1, \cdots, \varphi_k$, refining the abstraction requires additional predicates. The refinement algorithm takes as arguments the original PVS assertion f, a new list of predicates $\varphi_{k+1}, \cdots, \varphi_l$, and a context Γ_α computed previously. The context Γ_α is a hash-table which associates to each atom the BDD representing its abstraction, that is the BDD α, and the set *fail* of BDDs. The refinement algorithm descends through the structure of f and refines each sub-formula with the new predicates. The refinement algorithm is similar to the algorithm computing $\alpha_+(P)$ of Figure 1. However the variables α and *fail* are initialized with their already computed values. This allows us to take advantage of the success or failure of already executed proofs. The new set of test points is defined as the disjunctions formed using the literals $B_{k+1}, \cdots, B_l$ and their negation. This set is augmented with the boolean expressions over the old variables $B_1, \cdots, B_k$ for which the proof previously failed. The algorithm returns a more precise approximation of P.

We implemented our abstraction and refinement algorithms as a proof strategy defining a semi-decision procedure that abstracts an original PVS formula and then applies model checking. If model checking fails, the abstraction is refined until model checking succeeds. This strategy is expressed as follows in the PVS strategies language

```
(TRY (THEN  (abstract (phi_1...phi_k)) (model-check))
            (skip)
            (REPEAT
             (LET ((Φ (new-list-of-predicates)))
             (THEN (refine Φ) (model-check)))))
```

Our refinement algorithm tries to eliminate as much of the nondeterminism created by the over-approximation of the transition relation as possible. Absence

of nondeterminism can be easily detected by checking that when the abstraction of a transition $\alpha_+(P(s_1, s_2), C)$ is computed, the index i will never reach a value greater than 1. For instance, the abstraction of the assertion

$$e(s_1, s_2) \equiv s_2 = s_1 \text{WITH } [\text{sem} := \text{sem}(s_1) + 1]$$

presented in Section 3 is nondeterministic since it contains the conjunct

$$(B_1(abs\_s_1) \Rightarrow (B_1(abs\_s_2) \vee B_2(abs\_s_2)).$$

Refining such an abstraction involves translating the predicate characterizing the next state, that is $(B_1(abs\_s_2) \vee B_2(abs\_s_2))$ into a disjunctive normal form. Then, for each disjunct, the pre-image is computed with respect the concrete assertion $e(s_1, s_2)$. In this particular case, the pre-images for $B_1(abs\_s_2)$ and $B_2(abs\_s_2)$ are, respectively, $\exists(s_2) : e(s, s_2) \wedge \varphi_1(s_2)$ and $\exists(s_2) : e(s, s_2) \wedge \varphi_2(s_2)$. Their simplified forms are respectively $sem(s) < 0$ and $sem(s) = 0$.

6 Conclusion

We have presented a general abstraction/refinement algorithm that preserves the full μ-calculus as the basis for an integration of abstract interpretation, model checking, and proof checking. We have implemented this boolean abstraction algorithm as an extension to the PVS theorem prover. This allows us to define powerful proof strategies combining deductive proof, induction, abstraction, and model checking within a single framework. It also allows our abstraction algorithm to be used in the framework of a richly expressive specification language encompassing finite, infinite-state, and parametric systems. The computation of the abstraction is completely automatic, and uses the PVS decision procedures to test the generated implications.

We are currently investigating cases where it is possible to detect whether a constructed abstraction *strongly* preserves fragments of the μ-calculus so that abstract counterexamples yield concrete ones. This is done by finding sufficient conditions allowing us to use the various preservation results presented in [LGS+95,DGG94].

The new PVS version includes code generation capabilities, and as future work, we plan to define abstraction construction in the PVS specification language, and to automatically extract the code implementing the abstraction operation. Such experiments are similar to the ones presented in [vHPPR98] where, for instance, the code implementing a BDD simplifier is extracted automatically from its formal specification.

References

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of the 9th Conference on Computer-Aided Verification, CAV'98*, LNCS. Springer Verlag, June 1998.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, January 1977.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CU98] Michael Colon and Thomas Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Proceedings of the 9th Conference on Computer-Aided Verification, CAV'98*, LNCS. Springer Verlag, June 1998.

[DGG94] D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: Abstractions preserving ∀CTL*, ∃CTL* and CTL*. In Ernst-Rudiger Olderog, editor, *IFIP Conference PROCOMET'94*, pages 561–581, 1994.

[GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification CAV'97*, LNCS 1254, Springer Verlag, 1997.

[HS96] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, number 1051 in Lecture Notes in Computer Science, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.

[LGS+95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design, Vol 6, Iss 1, January 1995*, 1995.

[OSRSC98] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *The PVS Specification Language.* Computer Science Laboratory, SRI International, Menlo Park, CA, August 1998.

[PH97] A. Pardo and G.D. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. In *Conference on Computer Aided Verification CAV'97*, LNCS 1254, Springer Verlag, 1997.

[RSS95] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model checking with automated proof checking. In *Computer-Aided Verification, CAV '95*, number 939 in Lecture Notes in Computer Science, Liège, Belgium, 1995. Springer-Verlag.

[Rus97] John Rushby. Specification, proof checking, and model checking for protocols and distributed systems with PVS. In *FORTE/PSTV '97*, Osaka, Japan, November 1997.

[vHPPR98] Friedrich von Henke, Stephan Pfab, Holger Pfeifer, and Harald Rueß. Case studies in meta-level theorem proving. In Jim Grundy and Malcolm Newey, editors, *Theorem Proving in Higher Order Logics: 11th International Conference, TPHOLs '98*, volume 1479 of *Lecture Notes in Computer Science*, pages 461–478, Canberra, Australia, September 1998. Springer-Verlag.

Lazy Compositional Verification*

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
shankar@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (415) 859-5272 Fax: +1 (415) 859-2844

Abstract. Existing methodologies for the verification of concurrent systems are effective for reasoning about global properties of small systems. For large systems, these approaches become expensive both in terms of computational and human effort. A *compositional* verification methodology can reduce the verification effort by allowing global system properties to be derived from local component properties. For this to work, each component must be viewed as an open system interacting with a well-behaved environment. Much of the emphasis in compositional verification has been on the *assume-guarantee* paradigm where component properties are verified contingent on properties that are assumed of the environment. We highlight an alternate paradigm called *lazy composition* where the component properties are proved by composing the component with an abstract environment. We present the main ideas underlying lazy composition along with illustrative examples, and contrast it with the assume-guarantee approach. The main advantage of lazy composition is that the proof that one component meets the expectations of the other components, can be delayed till sufficient detail has been added to the design.

1 Introduction

In the last two decades, there has been considerable progress in the verification of concurrent, reactive systems. Much of the research has been devoted to the development of formalisms such as temporal logics [Eme90,Lam94,MP92,CM88] and

* Supported by the Air Force Office of Scientific Research under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931 and CCR-9300444. Based on earlier work [Sha93b] funded by Naval Research Laboratory (NRL) under contract N00015-92-C-2177. Connie Heitmeyer, Ralph Jeffords, and Pierre Collette gave useful feeback on the work cited above. John Rushby, Sam Owre, and Nikolaj Bjørner provided detailed comments on drafts of this paper. Presentations of earlier versions of this work at the meetings of IFIP Working Group 2.3 and at COMPOS'97 yielded valuable insights and criticisms. Martín Abadi and Leslie Lamport prompt and helpful in their responses to various technical queries and with feedback on earlier drafts.

From COMPOS'97: Springer LNCS Vol. 1536, pp. 541-564

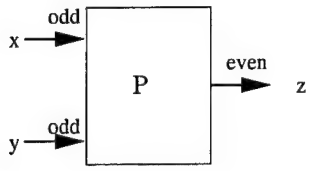


Fig. 1. Even number generator

process algebras [Hoa85,Mil80], and verification methods [Bar85,dBdRR90,dBdRR94,Sha93a] based on deduction [Eme90, Lam94, MP92, CM88] and model checking [CES86, Kur93, Hol91]. While these techniques are effective on small examples—mutual exclusion, basic cache consistency algorithms, and simple communication protocols—the difficult problem of scaling these techniques up to large and realistic systems has remained largely unsolved.

Large-scale concurrent systems are usually defined by composing together a number of components or subsystems. The typical verification methods are non-compositional and require a global examination of the entire system. In the *deductive* approach to verification, this means that a property such as an invariant has to be verified with respect to each transition of all of the components in the system. Verification approaches based on *model checking* also fail to scale up gracefully since the global state space that has to be explored can grow exponentially in the number of components [GL94]. The purpose of a *compositional* verification approach is therefore to shift the burden of verification from the global level to the local, component level so that global properties are established by composing together independently verified component properties.

To motivate compositional verification, we can consider a very simple example of an adder component P shown in Figure 1 that adds two input numbers x and y and places the output in z. Here x, y, and z can be program variables, signals, or latches depending on the chosen model of computation. The system containing P as a component might require its output z to be an even number, but obviously P cannot unconditionally guarantee this property of the output z. It might be reasonable to assume that the environment always provides odd number inputs at x and y, so that with this assumption it is easy to show that the output numbers at z are always even. Only local reasoning in terms of P is needed to establish that z is always even when given odd number inputs at x and y.

If, as is shown in Figure 2, P is now composed with another component Q that generates the inputs at x and y, then to preserve the property that only even numbers are output at z, Q must be shown to output only odd numbers at x and y. However, the demonstration that Q provides only odd numbers as outputs at x and y might require assumptions on the inputs taken by Q, where z itself might be such an input. If in showing that Q produces odd outputs at x and

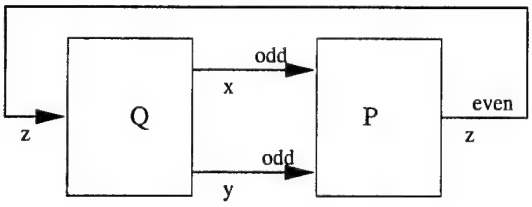


Fig. 2. Odd and even number generators

y, one has to assume that the z input is always even, then we have an obvious circularity and nothing can be concluded about the oddness or evenness of x, y, and z. If this circularity can somehow be broken, we then have a form of well-founded mutual recursion between P and Q that admits a proof by simultaneous induction that x and y are always odd and z is always even. The circularity can be broken by noting that that a z output for P is even as long as the preceding x and y inputs are odd, and the x and y outputs for Q are odd as long as the preceding z input is even.

The assume-guarantee paradigm is the best studied approach to compositional verification [AL93, AL95, AP93, CMP94, Col93, Hoo91, Jon83, MC81, PJ91, Pnu84, Sta85, XCC94, XdRH97, Zwi89]. In this approach, a property of a component is stated as a pair (A, C) consisting of a guarantee property C that the component will satisfy provided the environment to the component satisfies the assumption property A. The interpretation of (A, C) has to be carefully defined to be non-circular. Informally, a component P satisfies (A, C) if the environment to P violates A before the component fails to satisfy C. When two or more components, P_1 satisfying (A_1, C_1) and P_2 satisfying (A_2, C_2), are composed into a larger component $P_1 \| P_2$, the assumption A together with property C_1 of component P_1 must be used to show that P_1 does not violate assumption A_2, and correspondingly, A and C_2 must be used to show that P_2 does not violate A_2. Discharging these proof obligations allows one to conclude that the composite component $P_1 \| P_2$ has a similar property (A, C) where C follows from A, C_1, and C_2. The assume-guarantee technique as described informally still suffers from the earlier circularity. The formal details of the assume-guarantee technique are deferred to Section 2. The assume-guarantee approach has been more widely studied than actually used. The primary difficulty in applying this approach for compositional verification is that it requires component guarantee properties to be strong enough to entail any potential environment constraints. It is obviously not easy to anticipate all the potential constraints that might be placed on a component by the other components in a system.

The *lazy composition* approach advocated in this paper builds on conventional techniques while avoiding the difficulties associated with the assume-guarantee approach [Sha93b]. Lazy composition works at the level of the *specification* of component behavior. In lazy composition, a property C of a component

specified as P is actually proved of the system $P\|E$ obtained by composing P with an abstract environment specification E that captures the expected behavior of the environment. When the component specification P is composed with another component specification Q, then C might no longer be a property of the specification $P\|Q$ since Q might not satisfy the constraint E. However, C is a property of the composition $P\|(Q \wedge E)$ obtained by strengthening Q to additionally satisfy E. This allows local properties such as C to be used as global properties of the specification of a larger system. If in fact the combined specification $P\|(Q \wedge E)$ can be simplified to $P\|Q$, then clearly the constraint E is redundant and can be eliminated. However, it is not imperative that (properties guaranteed by) Q already imply E as is the case with the assume-guarantee technique. While the assumed environment specification has eventually to be shown to hold of the other components in the system, this proof obligation can be discharged lazily as the system design is being refined. The demonstration that $P\|(Q \wedge E)$ is refined by $P\|Q$ uses inductive reasoning on computations so that any possible circularity between assumptions E and guarantees C is avoided. Thus lazy composition allows global properties to be proved by local component-wise reasoning combined with a one-time demonstration that each component satisfies the accumulated constraints imposed by the other components. There are several other tradeoffs between lazy composition and assume-guarantee reasoning that are discussed in Section 3.

The lazy composition approach is quite general and can be applied to a wide variety of synchronous and asynchronous computational models, but this paper considers only one such model, namely, asynchronous transition systems with interleaving composition.

We first present some background on compositional verification in Section 2. Lazy composition is introduced in Section 3. Some examples illustrating the use of lazy composition in verifying safety properties are presented in Section 4. The elimination of environment constraints by means of refinement proofs is described in Section 5. The verification of liveness properties using lazy composition is given in Section 6. A comparison between lazy composition and other compositional approaches is given in Section 7.

2 Background

The presentation in this paper is entirely at the semantic level where we are dealing with states, predicates (sets) and relations on states, computations as infinite sequences of states, and properties as sets of computations. We will also speak of sets of sequences and properties interchangeably.

Asynchronous Transition Systems. In its simplest form, an *asynchronous transition system* is a triple $\langle \Sigma; I, N \rangle$ of a state type Σ, an initial set of states I, and a reflexive (stuttering-closed) *next-state* relation N that defines the possible

atomic actions of the system. Seen as a *closed system*, i.e., one with no interaction with an outside environment,[2] a valid *computation* of such a system consists of an infinite sequence of states σ whose initial state $\sigma(0)$ is in I, i.e., $I(\sigma(0))$ holds, and N holds of each pair of adjacent states, i.e., for all i, $N(\sigma(i), \sigma(i+1))$. A property is a set of infinite state sequences. If P is an asynchronous transition system, the set of its computations in the closed interpretation is represented as $[\![P]\!]$. The transition system P *has a property* A, in symbols, $[\![P]\!] \models A$, iff the set of computations $[\![P]\!]$ is a subset of the set of sequences corresponding to the property A. We write $\models A$ when the property A is valid, i.e., contains all the infinite sequences. Properties (sets of infinite sequences) can be combined with connectives $\neg A$ (complement), $A \vee B$ (union), $A \wedge B$ (intersection), and $A \supset B$ which is defined as $\neg A \vee B$. One transition system P *refines* another transition system Q when $\models [\![P]\!] \supset [\![Q]\!]$. In typical usage below, a transition system will be given as $\langle I, N \rangle$ leaving the state type Σ implicit.

Safety Properties. A safety property informally asserts that nothing bad happens during a computation. Let $\sigma[i]$ represent the finite prefix consisting of the first i states $\sigma(0) \ldots \sigma(i-1)$ of σ. A safety property [AS85] is one that excludes an infinite sequence σ exactly when it excludes all extensions $\sigma[i] \circ \rho$ of some finite prefix $\sigma[i]$ of σ. This means that safety properties are falsified by some finite prefix of a sequence. For any property A, there is a property A^S (the *safety closure* of A) which is the strongest safety property containing A defined as $\{\sigma \mid \forall i : \exists \rho : \sigma[i] \circ \rho \in A\}$. The property (set) A^S is clearly a safety property. If A is a safety property, we say that $\sigma[n] \in A$ when $\sigma[n] \circ \rho \in A$ for some ρ.

Liveness Properties. Liveness properties assert that something good eventually happens during the computation. Such properties hold of some infinite extension of any finite sequence α, i.e, they can always be satisfied by an appropriately chosen sequence of states. A liveness property can exclude an infinite sequence σ but must contain some extension of $\sigma[i]$ for each i. Given a property A, let A^L (the *liveness closure* of A) be $A \vee \neg A^S$, where $\neg A^S$ represents the complement of A^S. Then A^L is a liveness property because if for some α there is no ρ such that $\alpha \circ \rho \in A^L$, then since $A \subseteq A^L$, $\forall \rho : \alpha \circ \rho \notin A$, but then $\forall \rho : \alpha \circ \rho \notin A^S$. This is a contradiction since every infinite sequence must be in A^L or A^S. Thus every property A can be expressed as the conjunction of a safety property A^S and a liveness property A^L [Sch87].

Stuttering Invariance. A set of sequences A is *stuttering invariant* if whenever $\sigma[i+1] \circ \rho \in A$ then $\sigma[i+1] \circ \sigma(i) \circ \rho \in A$. In words, if A contains a sequence, then it contains all variants of this sequence obtained by stuttering individual states in the sequence finitely often. Stuttering arises naturally when there is a

[2] The closed interpretation here means that each transition of a valid computation satisfies the next-state relation N leaving no room for any environment transitions other than those already specified by N.

notion of an observation of a transition system so that some of the transitions have no observable effect. Stuttering invariance is often imposed as a constraint on the allowable properties so that the resulting transition system can always be implemented using internal unobservable state components.

Published explanations of assume-guarantee proof techniques often implicitly rely on stuttering invariance without explicitly mentioning it. Stuttering invariance is needed to argue that if we are given safety properties A and B such that $\sigma[i] \in A$ and $\sigma[i] \in B$, then $\sigma[i] \in A \wedge B$. Such a result is valid if A and B are stuttering invariant properties. To see how the result can fail to hold, let A consist of the strictly increasing sequences of even numbers and B consist of the strictly increasing sequences of prime numbers. Both A and B are safety properties that are not stuttering invariant. The singleton prefix $\langle 2 \rangle$ is in both A and B but $A \wedge B$ is empty.[3]

Expressing Properties. The above notions of computation and property are typical of the use of linear-time temporal logics for stating and proving properties of closed systems. Examples of such logics include

- Manna and Pnueli's LTL [MP92] with the temporal operators $\bigcirc$ (next-time), $\Box$ (always), and $\Diamond$ (eventually). Properties expressed in LTL that use the $\bigcirc$ operator are not necessarily stuttering invariant.
- Chandy and Misra's Unity [CM88] with operators **invariant**, **stable**, **unless**, **until**, and **leadsto** which are applied to state predicates so that temporal formulas are not nested. Unity properties are stuttering invariant.
- Lamport's temporal logic of actions [Lam94] which drops the next-time operator from linear-time temporal logic but allows temporal operators to range over *actions*, i.e., binary relations over states. TLA is designed to admit only stuttering invariant properties.

In the examples below, we restrict ourselves to some simple operators for defining properties. If p is a predicate on states, then

1. **invariant** p holds of σ iff $\forall i : p(\sigma(i))$. This is a safety property.
2. **eventually** p holds of σ iff $\exists i : p(\sigma(i))$. This is a liveness property for any satisfiable predicate p since any finite sequence can be extended to one in which p eventually holds.

For a given transition system $\langle I, N \rangle$, the invariance of p can be proved using induction by showing that for all states s in Σ, $\vdash I(s) \supset p(s)$, and for all states s and s', and $\vdash p(s) \wedge N(s, s') \supset p(s')$.

[3] A weaker requirement than stuttering invariance suffices for the soundness of the assume-guarantee proof reasoning methods. A safety property A must include the infinite sequence $\sigma[i+1] \circ \sigma(i)^\omega$ obtained by infinitely stuttering the last state of any nonempty finite prefix $\sigma[i+1]$ in A.

Components as Open Systems. The next step is to extend the model to open systems so that components can be independently specified and composed to form larger systems. If Σ is the set of global states of the large system, then a component i can be given as a triple $\langle \Sigma; I_i, N_i \rangle$. However, we can no longer take the closed interpretation since a computation must include the actions taken by other components. In the *open system* interpretation, a computation is an infinite sequence of states whose initial state is in I_i and each pair of adjacent states is either related by N_i or is an arbitrary environment transition. The open system interpretation is much too liberal and does not admit any interesting properties since there are no constraints on the environment actions. This can be partially overcome by placing weak constraints on the environment actions, e.g., the values of the local variables of a component must be left unchanged by its environment. With some constraint on the environment actions, one can actually verify reasonably interesting local properties of a component. For example, in TLA [Lam94], the next-state relation of a component is written as $[N]_f$ which holds of a pair of states s, s' when $N(s, s') \vee f(s') = f(s)$. The state function f typically projects out the local variables of the component so that the environment transitions must not affect the values of these variables. In Lynch and Tuttle's I/O automata [LT87], a component is an input-enabled automaton with its own local state so that any component properties established with respect to this interpretation remain globally valid even in composition with other components.

Even with such restrictions on the environment behavior, the open system interpretation is somewhat weak since many properties of a component can only be proved by assuming a stronger degree of cooperation from the environment. We have already seen the example of the adder component of Figure 1 which can be shown to always output even numbers when given odd number inputs by its environment.

The Owicki–Gries Method. The Owicki–Gries method [OG76] is the first attempt at a component-wise decomposition of the verification problem. In this method, one proves a global invariant of the composition $P_1 \| P_2$ by showing it to be a local invariant of one of the components, say P_1, and a stable predicate, i.e., one that is never falsified, of the other component P_2. In other words, one component establishes the invariant and the other component does not falsify it. This method is not really compositional since it requires global reasoning on all the actions of each component in order to establish an invariant. The Owicki–Gries method was originally proposed in the framework of a proof-outline logic where program components are annotated with assertions. Such program-based proof methods can be quite restrictive when compared to the use of high-level behavioral specifications as given by asynchronous transition systems.

Compositional Verification Using the Assume-Guarantee Approach. The assume-guarantee approach originally proposed by Jones [Jon83] and Misra and Chandy [MC81] is perhaps the most widely studied compositional verification

technique for concurrent systems. The presentation of this approach given below is adapted from Abadi, Lamport, and Plotkin [AL93, AL95, AP93] and Collette [Col94]. An assume-guarantee specification of a a component property is given as a pair (A, C) consisting of an assumption property A and a guarantee property C. To capture (A, C) is defined as $A \xrightarrow{+}\triangleright C$ (A *secures* C) which is the subset of $A \supset C$ defined as $\{\sigma \in A \supset C \mid \forall i : \sigma[i] \in A^S \supset \sigma[i+1] \in C^S\}$. Thus $A \xrightarrow{+}\triangleright C$ rules out unrealizable implementations of $A \supset C$ that exhibit computations where C^S fails before the failure of A^S can be detected by the component. Similarly, $A \longrightarrow\triangleright C$ (A *maintains* C) is the set of σ in $A \supset C$ such that for all i, $\sigma[i] \in A \supset \sigma[i] \in C$. Note that $A \xrightarrow{+}\triangleright C \equiv (A \supset C) \wedge (A^S \xrightarrow{+}\triangleright C^S)$, and $A \longrightarrow\triangleright C \equiv (A \supset C) \wedge (A^S \longrightarrow\triangleright C^S)$.

Composition of components $P_1 \| P_2$ is defined so that $[\![P_1 \| P_2]\!]$ is the intersection of $[\![P_1]\!]$ and $[\![P_2]\!]$. Since P_1 and P_2 are specified to allow environment transitions, the composition of P_1 and P_2 includes all the interleavings of P_1 and P_2 actions, but also contains computations with simultaneous P_1 and P_2 actions.[4]

The main compositionality rule in the assume-guarantee method [AL95] is stated in Theorem 1.

Theorem 1.

$$\frac{\begin{array}{l} P_i \models A_i \xrightarrow{+}\triangleright C_i, \text{ for } i = 1, 2 \\ \models A^S \wedge C_1^S \wedge C_2^S \supset A_1 \wedge A_2 \\ \models A \xrightarrow{+}\triangleright (C_1 \wedge C_2 \longrightarrow\triangleright C) \end{array}}{P_1 \| P_2 \models A \xrightarrow{+}\triangleright C.}$$

In words, in order to show that the composition $P_1 \| P_2$ has property $A \xrightarrow{+}\triangleright C$, it suffices to establish the following premises of the compositionality rule:

1. Each P_i has property $A_i \xrightarrow{+}\triangleright C_i$.
2. The individual environment constraints A_1 and A_2 must be satisfied by the conjunction of the safety parts of the joint environment constraint A and the guarantee properties C_1 and C_2.
3. The joint commitment C must be maintained by the individual commitments C_1 and C_2 when secured by the environment assumption A.

The formal details justifying the assume-guarantee rule are fairly elaborate, but we can briefly convey some of the intuition by sketching the soundness

[4] To obtain a strict interleaving of P_1 and P_2 actions, such joint actions can be excluded by asserting that the variables written by P_1 and P_2 must be disjoint and never simultaneously updated. Another approach is to label each transition with the agent associated with it, and to have a disjoint set of agents associated with components P_1 and P_2.

argument. It is sufficient to focus our attention on infinite sequences σ such that $\sigma \in (A_1 \xrightarrow{+} C_1) \wedge (A_2 \xrightarrow{+} C_2)$. To show $\sigma \in A \xrightarrow{+} C$, we need to prove both $\sigma \in A \supset C$ and $\sigma \in A^S \xrightarrow{+} C^S$. The argument proceeds in three steps:

- $\sigma \in (A^S \xrightarrow{+} C_1^S \wedge C_2^S)$.
  That is, for any n, $\sigma[n] \in A^S$ implies $\sigma[n+1] \in C_1^S \wedge C_2^S$. This can be proved by induction on n using premises 1 and 2 while noting that the stuttering invariance of A^S, C_1^S, and C_2^S is used in this argument.
- $\sigma \in A^S \xrightarrow{+} C^S$.
  By premise 3, for any n, $\sigma[n] \in A^S$ implies $\sigma[n+1] \in C_1^S \wedge C_2^S \longrightarrow C^S$. From step 1, we therefore have $\sigma[n+1] \in C^S$.
- $\sigma \in A \supset C$. For $\sigma \in A$, since $A \supset A^S$, we have by $A^S \xrightarrow{+} C_1^S \wedge C_2^S$ that $\sigma \in C_1^S \wedge C_2^S$. By premise 2, this yields $\sigma \in A_1 \wedge A_2$. By premise 1 and the definition of $\xrightarrow{+}$, we have that $\sigma \in C_1 \wedge C_2$. We can then apply premise 3 with the definitions of the connectives $\xrightarrow{+}$ and $\longrightarrow$ to obtain $\sigma \in C$.

There are some approaches to modular verification based on model checking that employ a weak form of assume-guarantee reasoning. In the work of Grumberg and Long [GL94], assume-guarantee properties (A, C) are treated as implications $A \supset C$ and not $A \xrightarrow{+} C$. Note that the use of implication for assume-guarantee reasoning is not valid in general, and is sound only for a restricted form of Theorem 1 where the cycle of dependencies between A_1, C_2, A_2, and C_1 has been broken. If A and C are just linear-time temporal logic (LTL) formulas, then LTL model checking can be used to verify $A \supset C$ of component P since this implication is also in LTL. If C is a CTL or CTL* formula, then the situation is more complicated since the implication $A \supset C$ is not a well-formed CTL or CTL* state formula, and furthermore, it does not capture the intended meaning of A as an assumption [Jos90] which is that C must hold on the computation tree whose paths have been pruned according to A. Then, A can be chosen as a $\forall$CTL formula that characterizes the subtree of the computation tree that meets the assumption. For the case of $\forall$CTL assumptions and synchronous Moore machine composition, Grumberg and Long give a way of compiling the assumption A into a tableau automaton A^T so that $P \| A^T \models C$ iff $P \models A \supset C$. Kupferman and Vardi [KV96] analyze the complexity of various linear and branching-time variants of modular model checking. Alur and Henzinger [AH96] give an assume-guarantee rule for proving language containment in the context of the synchronous composition of a form of Mealy machines called *reactive modules*.

3 Lazy Composition

Lazy composition differs from the assume-guarantee approach in several respects.

1. *Components are not treated as blackboxes.* Compositional verification merely requires that properties be proved locally at the component level. It does not require that components be treated as blackboxes for this purpose. The assume-guarantee approach requires the assumptions to be discharged solely by means of the guarantee properties of a component. The actual implementation of the component is never used for discharging proof obligations. This means that the guarantee properties must either somehow anticipate the possible constraints imposed by other components, or they must contain implementation details. Lazy composition on the other hand does not take a blackbox view of components and allows the behavioral specification to be used for discharging the constraints imposed by other components. Since a typical high-level behavioral specification might not contain enough detail to discharge such external constraints, lazy composition allows the constraints to be discharged lazily as the specification is being refined.
   Blackbox assume-guarantee specifications can be independently refined to yield implementations in terms of smaller blackbox components. Abadi and Lamport [AL95] give a *decomposition* rule for showing that $P'\|Q'$ refines $P\|Q$ when P' refines P and Q' refines Q. This rule has a premise similar to premise 2 of the compositionality rule of Theorem 1 which has the same drawback of requiring the environment constraints to be anticipated in the blackbox specification.
2. *Composition is not necessarily conjunction.* Conjunction can be used to define the interleaving composition of two asynchronous transition systems by a suitably chosen global constraint (see [AL95] and Footnote 4). Instead of encoding composition using conjunction, we regard the definition of the precise notion of composition as something that is fixed by the model of computation and not by the inference rule for composition. For asynchronous composition, one takes the interleaving of the atomic actions of each component, whereas for synchronous composition, i.e., globally clocked systems, one takes the conjunction of the atomic actions. Other formalisms that have asynchronously operating components with synchronous communication, e.g., CSP [Hoa85], can be modelled by means of a suitable definition of composition.
3. *Environment assumptions are specified as abstract components not properties.* One difficulty with environment assumptions as properties is that they apply to both the environment and the component. Typically, these constraints should apply only to environment actions and not the component actions. If we take the example of a bank account component, the environment might be required to only deposit and not withdraw money from the component but such a constraint should not apply to the component. There is no elegant way of stating this distinction between the component and its environment when the environment constraints are stated as properties rather than abstract components.
4. *No assume-guarantee proof obligations are generated.* With lazy composition, properties of a component P_i are proved in the context of an abstract

environment E_i. The composition rule ensures that all local properties are global properties of the composition. It does this by adding (conjoining, as explained below) the environment constraints of one component to the specification of the other component so that the resulting system has the form $(P_1 \wedge E_2) \| (P_2 \wedge E_1)$.

This form of composition appears dishonest (and lazy) since it sidesteps the question of whether the original specification of one component satisfies the environment assumptions of the other. However, specifications are meant to be partial and are therefore not always strong enough to anticipate the environment constraints that can be placed on a component. The best that one can do therefore is assert that if the specifications of each component is strengthened with the environment constraints required by the other component, then the resulting system satisfies the local properties of both components. If a component specification is strong enough to discharge any constraints placed on it, then the strengthening is redundant and can be eliminated by simplification. Otherwise, an implementation of the component is required to satisfy the stronger specification including the environment constraint.

The flexibility in postponing the assume-guarantee proof obligations is needed since some proofs might require the additional information that is provided when the specification is refined. If these proof obligations have to be proved as in the assume-guarantee proof method, then the component specifications must be quite detailed and strong. Since no proof obligations are discharged and environment assumptions impose additional, possibly unanticipated, constraints on a component specification, a component cannot be independently refined in the lazy composition approach. A component can only be independently refined when the component specification already implies all the environment constraints that might be required of it. There is, however, an advantage to refining in the global context where these assumptions are known since global properties can be exploited in the refinement (see Section 5).

5. *Composition can yield inconsistent specifications.* This is also the case when composition is defined as conjunction. In the case of lazy composition, this can arise because there is no computation that is compatible with the collection of constraints given in the specification.

Summarizing the discussion so far, lazy composition takes the middle ground between global verification as used in the Owicki–Gries approach and the strictly modular, property-based verification used in the assume-guarantee approach. Lazy composition is a proof style that uses a suitably weak characterization of a cooperative environment in composition with which a component can exhibit a given property. Once such an environment has been identified, the familiar verification techniques for proving safety, liveness, and refinement properties can be used. In the presentation of lazy composition, it will be assumed for convenience that there is a fixed environment specification for each component,

but in practice, the environment can be varied according to the desired property of the component.

We now move on to the details of lazy composition for asynchronous transition systems while noting that the techniques can easily be adapted to other models and notions of composition. As already stated, an asynchronous transition system is given by a triple $\langle \Sigma; I, N \rangle$ consisting of the state Σ, an initialization predicate I on the state, and a binary next-state relation N. Given such a triple P of the form $\langle \Sigma; I, N \rangle$, the closed interpretation of P is written as $[\![P]\!]$ and defined as the set sequences $\{\sigma | I(\sigma(0)) \wedge \forall i : N(\sigma(i), \sigma(i+1))\}$. We focus mainly on closed interpretations since one cannot prove interesting properties of computations that admit arbitrary environment actions. When we are talking about components, we will assume that Σ is the global state type and omit it from the transition system.

Given two transition systems, P_1 of the form $\langle I_1, N_1 \rangle$, and P_2 of the form $\langle I_2, N_2 \rangle$, the composition $P_1 \| P_2$ is the transition system $\langle I_1 \wedge I_2, N_1 \vee N_2 \rangle$. Note that composition essentially yields the interleaving of the component transitions.

The environment E is also given as a transition system $\langle I^e, N^e \rangle$. A component together with its environment is given as a pair $P//E$. The set of computations corresponding to $P//E$, i.e., $[\![P//E]\!]$, is defined as $[\![P \| E]\!]$, i.e., the closed interpretation of $P \| E$. Note that though $P//E$ and $P \| E$ have the same computations, the notation $P//E$ is chosen to emphasize the syntactic asymmetry between component P and environment E.

Given two transition systems P_1 and P_2, the conjunction of these, $P_1 \wedge P_2$, is $\langle I_1 \wedge I_2, N_1 \wedge N_2 \rangle$. Let P_i^e denote the component-environment specification $P_i//E_i$. Given two component–environment specifications P_1^e and P_2^e, the *closed co-imposition* of these two specifications $P_1^e \otimes P_2^e$ is defined as the transition system $(P_1 \wedge E_2) \| (P_2 \wedge E_1)$. The *open co-imposition* of P_1^e and P_2^e, written as $P_1^e \times P_2^e$, is defined as $(P_1^e \otimes P_2^e)//(E_1 \wedge E_2)$ and its computations contain actions corresponding to

1. P_1 but respecting E_2,
2. P_2 but respecting E_1, and
3. Environment actions respecting E_1 and E_2.

The closed co-imposition $P_1^e \otimes P_2^e$ yields a system with only the actions of P_1 and P_2, whereas the open co-imposition $P_1^e \times P_2^e$ yields a system with environment actions that are constrained to conform to both E_1 and E_2. Both operators are associative and commutative. It is easy to see that the property preservation result given in Theorem 2 holds so that $[\![P_1^e \otimes P_2^e]\!]$ and $[\![P_1^e \times P_2^e]\!]$ are both subsets of $[\![P_1^e]\!]$, and hence any properties of P_1^e are also properties of $P_1^e \otimes P_2^e$ and $P_1^e \times P_2^e$.

Theorem 2. *1.* $\models [\![P_1^e \otimes P_2^e]\!] \supset [\![P_1^e]\!]$
2. $\models [\![P_1^e \times P_2^e]\!] \supset [\![P_1^e]\!]$

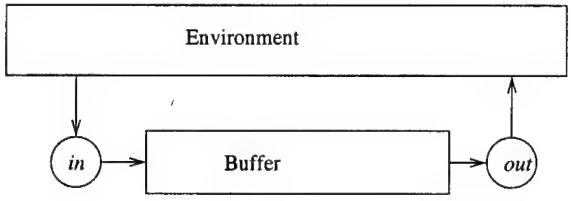


Fig. 3. A FIFO buffer with environment

We will henceforth ignore the closed co-imposition operator since its properties are similar to those of open co-imposition. The use of the co-imposition operation in lazy composition will be illustrated in Section 4. The obvious problem with lazy composition is that it asserts the property preservation of $P_1^e \times P_2^e$ and says nothing about $P_1 \| P_2$. By discharging proof obligations similar to those in Theorem 1, we can show that the transition system specification $P_1^e \times P_2^e$ is equivalent to the specification $(P_1 \| P_2) // (E_1 \wedge E_2)$, where the latter system contains actions corresponding to P_1 and P_2 without any restrictions, and the environment action $E_1 \wedge E_2$. In Section 5, we show that the environment constraints can be discharged in this manner by showing that P_2 refines E_1, and P_1 refines E_2. These refinement proofs can actually be carried out in the context of global invariants, i.e., invariants of $P_1^e \times P_2^e$. The resulting refinement proof obligations are similar to the assume-guarantee proof rule where $A^S \wedge C_1^S \wedge C_2^S$ must entail $A_1 \wedge A_2$.

4 Using Lazy Composition

Lazy composition will be illustrated by means of the example of a FIFO buffer component that is composed from two smaller FIFO buffer components. This examples has been frequently used with minor variations in the compositionality literature [Col93, AL95].

A single (bounded or unbounded) FIFO buffer component shown in Figure 3 consists of a buffer variable b that contains a queue of values, and the input and output variables *in* and *out* which contain values or are empty, i.e., contain a distinguished value $\perp$. Two history variables are used to specify the correct behavior of the buffer. The variable *inh* is a stack of all the non-$\perp$ values placed by the environment into *in*, and the variable *outh* is the stack of non-$\perp$ values read by the environment from *out*. The non-stuttering actions of the buffer are:

- Read a non-$\perp$ value from the variable *in* and enqueue it at the back of b while setting *in* to $\perp$. Formally, this is captured by the relation between the

pre-state $\langle in, b, out, inh, outh \rangle$ and the post-state $\langle in', b', out', inh', outh' \rangle$ as

$$read \triangleq \begin{cases} in \neq \bot \\ \wedge\ b' = enqueue(in, b) \\ \wedge\ in' = \bot \\ \wedge\ out' = out \\ \wedge\ outh' = outh \\ \wedge\ inh' = inh \end{cases}$$

– Dequeue a value from the front of queue b and place this value in the variable out when out is empty. Formally,

$$write \triangleq \begin{cases} nonempty?(b) \\ \wedge\ out = \bot \\ \wedge\ b' = dequeue(b) \\ \wedge\ out' = front(b) \\ \wedge\ outh' = push(front(b), outh) \\ \wedge\ in' = in \\ \wedge\ inh' = inh \end{cases}$$

In the initial state, all the variables associated with the buffer are empty. Formally,

$$init_b \triangleq (out = \bot \wedge b = outh = null).$$

The buffer component P is then given by the pair $\langle init_b, read \vee write \rangle$.

The environment component initializes the variables in and inh so that they are both empty:

$$init_e \triangleq (in = \bot \wedge inh = null).$$

In each non-stuttering action, the environment leaves b unchanged and may change the value of in when empty and may set the value of out to $\bot$. Formally,

$$load \triangleq (in = \bot \wedge in' \neq \bot \wedge inh' = push(in', inh))$$

$$unload \triangleq (out \neq \bot \wedge out' = \bot \wedge outh' = outh)$$

$$env \triangleq \begin{cases} (load \vee (in' = in \wedge inh' = inh)) \\ \wedge\ (unload \vee (out' = out \wedge outh' = outh)) \\ \wedge\ b' = b \end{cases}$$

The environment component E is given by the pair $\langle init_e, env \rangle$.

It is easy to prove by induction that

$$[\![P // E]\!] \models \textbf{invariant}\ inh = \overline{in} \circ q2s(b) \circ outh,$$

where $\circ$ is stack concatenation, $q2s(b)$ converts the queue b into a stack by repeatedly pushing elements from the front of queue b, and $\overline{in}$ is $push(in, empty)$ when $in \neq \bot$, and $empty$, otherwise. We have thus proved an invariant of a buffer component P by assuming that the environment behavior is as specified by E.

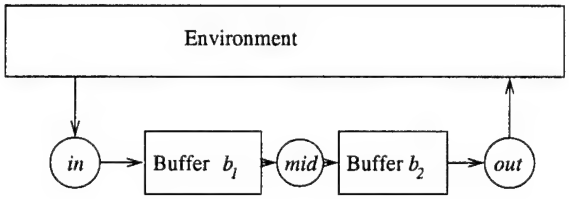


Fig. 4. FIFO buffer composed from smaller buffers

Compositional reasoning is used when two such buffers are composed as shown in Figure 4 to implement a single buffer. We do this by taking one instance $P_1//E_1$ of the buffer as specified above but renaming the variables b to b_1, *out* to *mid*, and *outh* to *midh*, and a second instance $P_2//E_2$ with b renamed to b_2, and where *in* and *inh* are just *mid* and *midh*, respectively. In other words, buffer P_1 communicates values to buffer P_2 via *mid*.

Having already proved the invariant above for a FIFO buffer P, the goal now is to prove a similar invariant $inh = \overline{in} \circ q2s(b) \circ outh$, for some b, for the composition $(P_1 \| P_2)//(E_1 \wedge E_2)$ of the two buffers. However, we cannot use the invariant proved of P for composite buffers with P_1 and P_2 since those invariants are proved for the systems $P_1//E_1$ and $P_2//E_2$.

The claim $\models [\![(P_1 \| P_2)//(E_1 \wedge E_2)]\!] \supset [\![P_1//E_1]\!]$ is not provable since the definitions of P_1 and P_2 are not strong enough to imply the constraints E_2 and E_1, respectively. This is because E_1 specifies that each environment action must leave the buffer variable b_1 unchanged and that the variable *in* must be written only by the environment. The actions of P_2 place no constraints on the update of the values of b_1 or *in*. Since we cannot demonstrate $\models [\![(P_1 \| P_2)//(E_1 \wedge E_2)]\!] \supset [\![P_1//E_1]\!]$, the invariant for P_1^e, namely, $inh = in \circ b_1 \circ midh$, cannot be used as a global invariant of $(P_1 \| P_2)//(E_1 \wedge E_2)$.

The best that we can do therefore is to conclude $\models [\![P_1^e \times P_2^e]\!] \supset [\![P_1^e]\!] \wedge [\![P_2^e]\!]$, so that the conjunction of the individual invariants holds for $P_1^e \times P_2^e$. From the conjunction of the two invariants:

1. $[\![P_1^e \times P_2^e]\!] \models$ **invariant** $inh = \overline{in} \circ q2s(b_1) \circ midh$
2. $[\![P_1^e \times P_2^e]\!] \models$ **invariant** $midh = \overline{mid} \circ q2s(b_2) \circ outh$

we can conclude

$$[\![P_1^e \times P_2^e]\!] \models \textbf{invariant } inh = \overline{in} \circ q2s(b_1) \circ q2s(mid) \circ q2s(b_2) \circ outh.$$

So if we take b to be $s2q(q2s(b_1) \circ \overline{mid} \circ q2s(b_2))$ where *s2q* is the inverse of *q2s* and converts a stack back into the corresponding queue, we have the desired invariant $inh = \overline{in} \circ q2s(b) \circ outh$ for $P_1^e \times P_2^e$.

In proving this invariant, we have used only the corresponding invariants of the component buffers and some elementary lemmas about the concatenation operation. We have not directly used the specification of individual buffers themselves. We have worked at the level of the *specification* of the behavior of the individual buffers rather than the corresponding *program* which would be a complete specification of each transition. Since specifications can be partial, it makes sense to conjoin the environment constraints to the component specification rather than discharge them as proof obligations. Thus a more detailed implementation will have to satisfy the higher-level specification of the component as well as the constraints on the component imposed by the other components in the combined system.

When refining P_1 to a more refined specification or a program in the context $P_1^e \times P_2^e$, it is valid to use all the global invariants that have been proved of $P_1^e \times P_2^e$. The introductory example involving odd and even numbers can be used to illustrate the use of such invariants in refinement. The system P there is of the form $\langle I_P, N_P \rangle$ where

$$I_P \triangleq even?(z)$$
$$N_P \triangleq (z' = x + y) \wedge (x' = x) \wedge (y' = y)$$

If P's environment constraint D is of the form $\langle I_D, N_D \rangle$ where

$$I_D \triangleq odd?(x) \wedge odd?(y)$$
$$N_D \triangleq odd?(x') \wedge odd?(y') \wedge z' = z$$

then we can prove the invariant $even?(z) \wedge odd?(x) \wedge odd?(y)$ for the system $P//D$. Let Q be defined to be $\langle I_Q, N_Q \rangle$ where

$$I_Q \triangleq odd?(x) \wedge odd?(y)$$
$$N_Q \triangleq (x' = x + z) \wedge (y' = y + z) \wedge (z' = z)$$

Let E be the unconstrained system consisting of the everywhere-true initialization predicate and next-state relation. We would now like to show that the constraint D is satisfied by Q, but this is not true in general. It does however hold in the context of the invariant $even?(z) \wedge odd?(x) \wedge odd?(y)$. The use of invariants allows $[\![(P//D) \times (Q//E)]\!]$ to be simplified to $[\![P \| Q \| D]\!]$ since $P \wedge E$ simplifies to P, $D \wedge E$ simplifies to D, and $Q \wedge D$ can be simplified to Q given

$$\vdash even?(z) \wedge odd?(x) \wedge odd?(y) \wedge N_Q \supset N_D.$$

We show how invariants can be used in proving a refinement relation between two transition systems using stepwise simulation in the next section.

5 Discharging Proof Obligations by Refinement

We now examine how the familiar notion of refinement via simulation can be used to simplify away the environment constraints E_1 and E_2 in the lazy composition $P_1^e \times P_2^e$. This is analogous to the assume-guarantee proof obligations (premise 2) except that lazy composition is more flexible about how and when these proof obligations are discharged. Recall that in the assume-guarantee approach, the assumptions of one component had to be discharged using the guarantee properties of all the components along with the global environment constraints. As we noted, this has the disadvantage that the guarantee properties have to be chosen to somehow anticipate the likely environment constraints. By contrast, in lazy composition, these proof obligations are discharged lazily during refinement.

The refinement rule establishes the conclusion $\models [\![P]\!] \supset [\![Q]\!]$ by showing that each transition of P can be simulated by a transition of Q . In particular, this means that P inherits all the properties of Q. The simulation of P transitions by Q transitions can be shown in the presence of invariants of P and Q. The invariants might be needed because the simulation relation between the actions of P and Q might not hold outside their respective reachable states. The invariant that is used for P can be an action invariant, a binary relation r on Σ such that $\forall i : r(\sigma(i), \sigma(i+1))$. In this case, we say that **invariant** r holds of σ. Given a state predicate p, an action r, and two transition systems P and Q of the form $\langle I_P, N_P \rangle$ and $\langle I_Q, N_Q \rangle$, respectively, the refinement rule is stated in Theorem 3.

Theorem 3.

$$\frac{\begin{array}{l}[\![P]\!] \models \textbf{invariant } r \\ [\![Q]\!] \models \textbf{invariant } p \\ \vdash p(s) \wedge r(s,s') \wedge N_P(s,s') \supset N_Q(s,s') \\ \vdash I_P(s) \supset I_Q(s)\end{array}}{\models [\![P]\!] \supset [\![Q]\!]}$$

The proof of the refinement rule is by a straightforward induction on the length of the computations in $[\![P]\!]$. The relevance of the refinement rule for compositional verification is that we can use it to eliminate the constraints imposed on one component by another. When composing specifications using the co-imposition operator, we end up with a specification $P_1^e \times P_2^e$ which is equivalent to $(P_1 \wedge E_2) \| (P_2 \wedge E_1) \| (E_1 \wedge E_2)$. To eliminate, say, E_2 from this specification, we need to show that $(P_1 \wedge E_2) \| (P_2 \wedge E_1) \| (E_1 \wedge E_2)$ can be refined by $P_1 \| (P_2 \wedge E_1) \| (E_1 \wedge E_2)$. The constraint E_1 can also be similarly eliminated. This kind of refinement can be carried out with the aid of a simple corollary to the refinement rule that can be used to show that $[\![P \| Q]\!]$ refines $[\![P \wedge E \| Q]\!]$ by showing that each P transitions can be simulated by an E transition.

Corollary 4.

$$\frac{\begin{array}{l}[\![(P \wedge E) \| Q]\!] \models \textbf{invariant } p \\ \vdash p(s) \wedge N_P(s,s') \supset N_E(s,s') \\ \vdash I_P(s) \supset I_E(s)\end{array}}{\models [\![P \| Q]\!] \supset [\![(P \wedge E) \| Q]\!]}$$

Note that any global invariant p can be used in proving the stepwise simulation. This is what justifies the use in Section 4 of the invariant $even?(z) \wedge odd?(x) \wedge odd?(y)$ in showing that the strengthening of the specification Q with D is redundant.

6 Liveness

Compositional liveness reasoning is needed for showing progress properties for a component contingent on similar progress properties of other components. For example, the FIFO buffer can only guarantee that an output will always eventually be written if the environment can guarantee that a value in the *out* variable will always eventually be read.

Liveness or progress assumptions have to be handled with some care in compositional verification. For example, suppose a component P guarantees that output z is eventually 4 assuming the input x is eventually 3, and conversely, component Q guarantees an eventual output 3 on x assuming that the input z is eventually 4. If the guarantee properties are used to discharge assumptions, then the composed system $P \| Q$ guarantees that z will eventually take on the value 4 and that eventually x will take on the value 3. This would be unsound since the system actually need not obey either eventuality for x or z and the individual assume-guarantee properties would still be satisfied. The assume-guarantee proof rule is carefully crafted to rule out this kind of circularity by ensuring in premise 2 that the assumptions have to be satisfied solely from the safety parts of the guarantee properties. Component liveness properties are instead expressed as implications in the property C_1 of a component P_1, where the antecedent of the implication is the fairness constraint on the other component. This antecedent is of course easily discharged in the conjunction $C_1 \wedge C_2$ if C_2 includes the fairness condition of P_2.

To admit proofs of liveness properties in lazy composition, it will be necessary to extend the notion of a transition system to include fairness conditions. An asynchronous transition system with fairness is of the form $\langle \Sigma; I, N, F \rangle$ where F is a *fairness* property that a valid computation must satisfy, i.e., $[\![\langle I, N, F \rangle]\!] \equiv [\![\langle I, N \rangle]\!] \wedge F$. It is desirable that the F component be used only to establish progress properties so that any safety property should follow from the system $\langle \Sigma; I, N \rangle$ without F. For this to be the case, the fairness condition F should be *machine closed*, i.e., any finite prefix $\sigma[n]$ in $\langle I, N \rangle$ should be extendable to a sequence $\sigma[n] \circ \rho$ in $[\![\langle I, N, F \rangle]\!]$. F is machine closed with respect to the transition system $[\![\langle I, N \rangle]\!]$ iff $[\![\langle I, N, F \rangle]\!]^S = [\![\langle I, N \rangle]\!]$. For example, if P is a transition system with only one state component x whose value is initially 0, and a next-state relation $x' = x + 2 \vee x' = x + 3 \vee x' = x$, then the property **eventually** $x = 3$ is not a machine-closed fairness condition since it excludes the computations in which x takes the value 2.

Typical notions of fairness such as weak and strong fairness are machine closed with respect to the closed interpretation of a single transition system.

An action r is said to be enabled in a state s, formally $enabled(r)(s)$, iff there exists a state s' such that $r(s, s')$ holds. A predicate p holds infinitely often on a sequence σ iff $\forall i : \exists j : j > i \wedge p(\sigma(j))$. Similarly, an action r holds infinitely often on σ iff $\forall i : \exists j : j > i \wedge r(\sigma(j), \sigma(j+1))$. A sequence σ is said to be *weakly fair* with respect to an action r iff either $\neg enabled(r)$ holds infinitely often or r holds infinitely often on σ. A sequence σ is said to be *strongly fair* with respect to action r iff r holds infinitely often on σ when $enabled(r)$ does. It can be shown that F is machine closed with respect to transition system $\langle I, N\rangle$ if F is a conjunction of weak and strong fairness assertions on actions $r_1, \ldots, r_n$ such that each r_i is *unblocked* in $\langle I, N\rangle$,[5] i.e.,

$$[\![\langle I, N\rangle]\!] \models \textbf{invariant}\ enabled(r_i) \supset enabled(r_i \wedge N).$$

When F is machine closed with respect to $\langle I, N\rangle$, we say that the fair transition system $\langle I, N, F\rangle$ is machine closed.

The situation is not so simple for transition systems whose computations include both component and environment transitions. The definition of composition for fair asynchronous transition systems is

$$\langle I_1, N_1, F_1\rangle \| \langle I_2, N_2, F_2\rangle \triangleq \langle I_1 \wedge I_2, N_1 \vee N_2, F_1 \wedge F_2\rangle.$$

The purpose of distributing the fairness conditions among the various components is to allow componentwise properties to be deduced using just the relevant global fairness conditions. In particular, machine closure is defined only with respect to a closed interpretation so that it is only required for specifications such as $P//E$ or $P_1^e \times P_2^e$. Given the above definition of composition for fair asynchronous transition systems, all fairness conditions are global and apply to all components.

In a blackbox style of component specification, implementability considerations require the component fairness condition to be machine closed with respect to the open interpretation, and also *receptive*, i.e., machine closed without relying on cooperation from the environment. The receptiveness constraint on the fairness condition can exclude unconditional strong fairness constraints since a hostile environment can enable and disable a component action r without allowing the component a chance to execute r. Receptiveness is a sensible restriction when specifying an open component operating in an uncontrolled environment, but this is not the situation in compositional verification since the environment includes components whose specifications are an integral part of the design.

Given the definition of composition extended with fairness conditions, the definitions of the operations $\otimes$ and $\times$ remain unchanged from Section 3. The property preservation results claimed in Theorem 2 also holds in the presence of fairness conditions.

[5] Abadi and Lamport [AL95] state this constraint differently by requiring each r_i to be a possible program action. This is equivalent since the fairness constraint r_i can just as well taken to be $N \wedge r_i$.

There is however one serious problem with lazy composition in the presence of fairness. The co-imposition $P_1^e \times P_2^e$ of machine-closed specifications $P_1//E_1$ and $P_2//E_2$ is not necessarily machine closed. The co-imposition contains the conjunctions $P_1 \wedge E_2$, $P_2 \wedge E_1$, and $E_1 \wedge E_2$. The conjunction of two transitions systems $\langle I_1, N_1, F_1\rangle \wedge \langle I_2, N_2, F_2\rangle$ is defined as $\langle I_1 \wedge I_2, N_1 \wedge N_2, F_1 \wedge F_2\rangle$. Since the actions of the conjoined transition system are specified by $N_1 \wedge N_2$ which is more constrained than either N_1 or N_2, the fairness condition $F_1 \wedge F_2$ might not be machine closed in the resulting transition system. For example, let $x' = x+1$ be a possible action of P_1 where P_1 initializes x to 0 and has no actions that decrement or reset x. Then it can be proved of $[\![P_1^e]\!]$ that if the increment action is weakly fair, then **eventually** $x = 3$. However, if E_2 in P_2^e requires that x not be incremented, then the set of computations $[\![P_1^e \times P_2^e]\!]$ is empty since the only possible computations are those where the value of x is never changed and these are ruled out by the weak fairness requirement on the increment action. Of course, the property **eventually** $x = 3$ is vacuously preserved in this case. Note that machine closure is violated in this example even if E_2 contains no fairness conditions simply because E_2 blocks a fair action of P_1.

There is therefore a proof obligation that the system $P_1^e \times P_2^e$ be shown to be machine closed. In the special case of fairness conditions that only contain weak and strong fairness assertions, this proof obligation can be discharged by showing that each fair action is unblocked in the combined system.

The notion of refinement used to eliminate environment constraints has to be extended to fair asynchronous transition systems. The goal is to show that $\models [\![\langle I_P, N_P, F_P\rangle]\!] \supset [\![\langle I_Q, N_Q, F_Q\rangle]\!]$. For this, we need to add one additional premise to the refinement rule in Section 5.

Theorem 5.

$$\frac{\begin{array}{l}[\![P]\!] \models \textbf{invariant } r \\ [\![Q]\!] \models \textbf{invariant } p \\ \vdash p(s) \wedge r(s,s') \wedge N_P(s,s') \supset N_Q(s,s') \\ \vdash I_P(s) \supset I_Q(s) \\ \models [\![\langle I_P, N_P, F_P\rangle]\!] \supset F_Q\end{array}}{\models [\![P]\!] \supset [\![Q]\!]}$$

The discharging of the new premise can require temporal reasoning. For the case of fairness conditions that are conjunctions of weak and strong fairness assertions, one can simply show that to any weakly fair action r_i in Q, there is a weakly or strongly fair action r'_j in P such that

$$[\![\langle I_P, N_P\rangle]\!] \models \textbf{invariant } r'_j \supset r_i$$

and

$$[\![\langle I_P, N_P\rangle]\!] \models \textbf{invariant } \mathit{enabled}(r_i) \supset \mathit{enabled}(r'_j \wedge N_P).$$

Similarly, to each strongly fair action in Q, there must be a corresponding strongly fair action in P.

Returning to the example of the FIFO buffer, if the actions *read* and *write* are weakly fair, and the *unload* action for the buffer environment is weakly fair, then in any fair computation of this transition system it is always the case that a state in which $x = in \neq \perp$ is eventually followed by (i.e., *leads to*, in the terminology of temporal logic) a state in which $out = x$.

7 Discussion

We have argued thus far that lazy composition is superior to the assume-guarantee method for compositional verification on the grounds that:

1. Lazy composition employs proof methods that are already familiar whereas the assume-guarantee proof rule is quite formidable.
2. Assume-guarantee methods require specifications that can anticipate future environment constraints.
3. The assume-guarantee assumptions apply to both component and environment and it is awkward to restrict these so that they only constrain the environment.
4. Assume-guarantee specifications are more appropriate for writing blackbox characterizations of open components rather than for compositional verification where the point is to achieve a useful decomposition of the verification task.

The advantage of lazy composition with respect to non-compositional, global reasoning as characterized by the Owicki–Gries approach [OG76] is that it combines the simplicity of global reasoning with the economy of using an abstract characterization of the environment rather than the actual components in the environment. This abstract characterization can be used to prove a number of component properties. The actual components can then be shown to conform to this abstract characterization by means of a refinement proof.

The Owicki–Gries approach is subsumed by lazy composition. If P is a component that is required to satisfy an invariant p, then we can take the environment E to be the transition system that merely preserves p, i.e., $\vdash N_E(s, s') \wedge p(s) \supset p(s')$. Then the refinement proof obligation reduces to a global demonstration that each component that is composed with P preserves the invariant. This is obviously the most general assumption one can make of an environment to P given that one wants to establish the invariant p, but it is not the optimal way to use lazy composition. The more appropriate use of lazy composition is by describing the allowed or intended environment actions that are relevant to the state variables that are read or written by component P and that are needed to obtain useful properties of P. Thus lazy composition modularizes the global reasoning by identifying suitable abstractions for the environment of each component.

7.1 Other Applications of Lazy Composition

We have employed lazy composition in the verification of the safety properties of an N-process mutual exclusion algorithm [Sha97] and the alternating-bit communication protocol [BSW69]. These verifications have been carried out using PVS [ORS92]. The mutual exclusion algorithm has been verified using a combination of induction, abstraction, and model checking. The algorithm uses a Boolean *turn* variable for each process to arbitrate access to successive rounds of competition using 2-process mutual exclusion, for eventual access to the critical section. The environment to each process has to be constrained to not affect the value of this *turn* variable in an undesirable way, e.g., when a process has checked the *turn* value and has entered its critical section.

The example of the alternating-bit protocol consists of a sender process, a receiver process, and the message and acknowledgement channels. The sender process constrains its environments merely to drop messages from the message channel, and the receiver process similarly constrains the value of the acknowledgement channel. With these constraints, it is possible to carry out a modular verification of the safety property of the alternating-bit protocol where all the invariants are proved solely by local reasoning in terms of the receiver or the sender process, possibly using previously proved global invariants.

8 Conclusions

We have presented the details of the paradigm of lazy compositional verification. This approach has several advantages over the assume-guarantee paradigm. We have formalized lazy composition verification within PVS [ORS92] and verified several medium-scale examples with this approach. We do not yet have any conclusive evidence that the method scales up to larger systems. Lazy composition can be adapted to models other than asynchronous transition systems by suitably altering the definitions of composition, conjunction, and refinement. Lazy composition does not need any new verification machinery since it builds on existing techniques for proving safety, liveness, and refinement properties.

References

[AH96] Rajeev Alur and Thomas A. Henzinger. Reactive modules. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 207–218, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[AL93] Martín Abadi and Leslie Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[AL95] Martín Abadi and Leslie Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[AP93] Martín Abadi and Gordon D. Plotkin. A logical view of composition. *Theoretical Computer Science*, 114(1):3–30, 1993.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[Bar85] H. Barringer. *A Survey of Verification Techniques for Parallel Programs*, volume 191 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[BSW69] K. A. Bartlett, R. A. Scantlebury, and P. T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Communications of the ACM*, 12(5):260, 261, May 1969.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.

[CMP94] Edward Chang, Zohar Manna, and Amir Pnueli. Compositional verification of real-time systems. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 458–465, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[Col93] P. Collette. Application of the composition principle to Unity-like specifications. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of TAPSOFT '93*, volume 668 of *Lecture Notes in Computer Science*, pages 230–242, Berlin, 1993. Springer-Verlag.

[Col94] Pierre Collette. An explanatory presentation of composition rules for assumption-commitment specifications. *Information Processing Letters*, 50(1):31–35, April 1994.

[dBdRR90] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, volume 430 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

[dBdRR94] J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors. *A Decade of Concurrency: Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*, Noordewijkerhout, The Netherlands, 1994. Springer Verlag.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 16, pages 995–1072. Elsevier and MIT press, Amsterdam, The Netherlands, and Cambridge, MA, 1990.

[GL94] Orna Grumberg and David E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1985.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[Hoo91] J. Hooman. *Specification and Compositional Verification of Real-Time Systems*, volume 558 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM TOPLAS*, 5(4):596–619, 1983.

[Jos90] B. Josko. Verifying the correctness of AADL modules using model checking. In de Bakker et al. [dBdRR90], pages 386–400.

[Kur93] R.P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes.* Princeton University Press, Princeton, NJ, 1993.

[KV96] O. Kupferman and M. Y. Vardi. Module checking. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification96*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86. Springer Verlag, 1996.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[LT87] N.A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth Annual Symposium on Principles of Distributed Computing, New York*, pages 137–151. ACM Press, 1987.

[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification.* Springer-Verlag, New York, NY, 1992.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[PJ91] P. K. Pandya and M. Joseph. P–A logic — a compositional proof system for distributed programs. *Distributed Computing*, 5(1):37–54, 1991.

[Pnu84] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Logic and Models of Concurrent Systems*, NATO-ASI, pages 123–144. Springer Verlag, 1984.

[Sch87] Fred B. Schneider. Decomposing properties into safety and liveness using predicate logic. Technical Report 87-874, Department of Computer Science, Cornell University, Ithaca, NY, October 1987.

[Sha93a] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3):225–262, September 1993.

[Sha93b] N. Shankar. A lazy approach to compositional verification. Technical Report SRI-CSL-93-8, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[Sha97] N. Shankar. Machine-assisted verification using theorem proving and model checking. In Manfred Broy and Birgit Scheider, editors, *Mathematical Methods in Program Development*, volume 158 of *NATO ASI Series F: Computer and Systems Science*, pages 499–528. Springer, 1997.

[Sta85] E. W. Stark. A proof technique for rely/guarantee properties. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 206 of *Lecture Notes in Computer Science*, pages 369–391. Springer Verlag, 1985.

[XCC94] Q.-W. Xu, A. Cau, and P. Collette. On unifying assumption–commitment style proof rules for concurrency. In B. Jonsson and J. Parrow, editors, *CONCUR'94*, volume 836 of *Lecture Notes in Computer Science*, pages 267–282. Springer Verlag, 1994.

[XdRH97] Q.-W. Xu, W.-P. de Roever, and J.-F. He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness*, volume 321 of *Lecture Notes in Computer Science*. Springer Verlag, 1989.

Fair Synchronous Transition Systems and their Liveness Proofs *

Amir Pnueli
Dept. of Applied Math. and CS
The Weizmann Institute of Science
Rehovot, ISRAEL

Natarajan Shankar Eli Singerman
Computer Science Laboratory
SRI International
Menlo Park CA, USA

Technical Report SRI-CSL-98-02

Abstract

We present a compositional semantics of synchronous systems that captures both safety and progress properties of such systems. The *fair synchronous transitions systems* (FSTS) model we introduce in this paper extends the basic αSTS model [KP96] by introducing operations for parallel composition, for the restriction of variables, and by addressing fairness. We introduce a weak fairness (justice) condition which ensures that any communication deadlock in a system can only occur through the need for external synchronization. We present an extended version of linear time temporal logic (ELTL) for expressing and proving safety and liveness properties of synchronous specifications, and provide a sound and compositional proof system for it.

*This research was supported in part by the Minerva Foundation, by an infrastructure grant from the Israeli Ministry of Science, by US National Science Foundation grants CCR-9509931 and CCR-9712383, and by US Air Force Office of Scientific Research Contract No. F49620-95-C0044. Part of this research was done as part of the European Community project SACRES (EP 20897). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, AFOSR, the European Union, or the U.S. Government. We are grateful to Sam Owre for lending assistance with PVS.

From FTRTFT'98: Springer LNCS Vol. 1486, pp. 198-209

1 Introduction

Synchronous languages are rapidly gaining popularity as a high-level programming paradigm for a variety of safety-critical and real-time control applications and for hardware/software co-design. Synchronous languages are used to define systems that are:

- Reactive: Engaged in an ongoing interaction with an environment.
- Event-triggered: System computation is triggered by the arrival and absence of external inputs.
- Concurrent: A system is composed of subsystems that are computing and interacting concurrently.
- Synchronous: The system response to input stimuli is "instantaneous".

The present paper presents a unifying transition system model for synchronous languages based on *fair synchronous transition systems* (FSTS) that can be used as a semantic basis for synchronous languages. Such a unifying semantic model can be used to specify the temporal behavior of synchronous systems and to relate different synchronous system descriptions.

There are two major classes of synchronous languages [H93]. The *imperative* languages like ESTEREL [BG92] and STATECHARTS [Har87], and *declarative* languages like LUSTRE [CHPP87] and SIGNAL [BGJ91]. The language ESTEREL has statements like:

- **present** *signal* **then** *statement* **else** *statement*: Execute **then** statement when *signal* is present, and **else** statement otherwise.
- **emit** *signal* : broadcast signal.
- **trap** *id* **in** ... **exit** *id* ...: A catch-throw mechanism for exception signaling and handling.
- **do** *statement* **watching** *signal*: preempt execution of *statement* when *signal* occurs.

These are combined with the usual control constructs like conditional branching and parallel composition.

Declarative languages like LUSTRE and SIGNAL express constraints on signal flows or infinite streams. In LUSTRE, each signal is defined in a mutually recursive way in terms of other signals. LUSTRE applies the various logical and arithmetic operators in a pointwise manner to signals so that `X + Y` is the signal that is the pointwise sum of signals `X` and `Y`. The basic operators for constructing expressions that define signals in LUSTRE are:

- **pre**(`X`) which is nil, $x_0, x_1, \ldots$, where `X` is $x_0, x_1, \ldots$.

- `X -> Y` which is the signal $x_0, y_1, y_2, \ldots$ where `X` is of the form $x_0, x_1, \ldots$, and `Y` is of the form $y_0, y_1, \ldots$.
- `X` **when** `B` which is the signal $\langle x_i \mid b_i \rangle$, i.e., the sequences $x_{i_1}, x_{i_2}, \ldots,$, where $i_1, i_2, \ldots$, are the positions at which the boolean signal `B` is true. The resulting clock of `X when B` is defined by the boolean signal `B`.
- **current** `X` which speeds up `X` to the next faster clock in terms of which `X` is defined. This is done by padding `X` with the previous x_i wherever `X` is undefined with respect to the faster clock.

For example,

`B`	T	F	T	F	F	T	T	F	F...
`X`	1	2	3	4	5	6	7	8	9...
`X` **when** `B`	1		3			6	7		...
current (`X` **when** `B`)	1	1	3	3	3	6	7	8	8...

LUSTRE programs are functional with respect to the input streams. All LUSTRE signals in a program are defined with respect to sub-clocks of a single master clock.

The language SIGNAL is the most liberal of the synchronous languages and is the primary motivation for the fair synchronous transition systems model presented here. SIGNAL is similar to LUSTRE but admits the specification of nondeterministic constraints and the signals are not all derived from a single master clock.

SIGNAL programs may also contain mutually recursive definitions of signals. Apart from the usual pointwise arithmetic, logical, and other data operators, the basic operators in SIGNAL are:

- `X $` **init** y_0 which is the signal having the same clock as `X` and the same signal as `X` but delayed by one clock tick and with initial value y_0.
- `X` **when** `B` is similar to the corresponding operator in LUSTRE but `X` and `B` can have different clocks. The clock of the resulting signal is the intersection of the two clocks and takes on the value of `X` whenever the boolean signal `B` is true.
- `X` **default** `Y` is the signal whose clock is the union of the clocks of `X` and `Y` and whose value is equal to the value of `X` when `X` is defined, and the value of `Y` otherwise.

SIGNAL programs can be combined by parallel composition `P||Q` which is the conjunction of the equations in `P` and `Q`. Hiding `P/X` is another operation where an externally visible signal `X` in `P` can be made local to a program `P/X`.

The features common to all these synchronous languages are that

- Signals are not persistent and can be present or absent in a computation state.
- State transitions can be governed by the presence of a signal in a state, its value when it is present, and its absence.
- A module specifies constraints on the signal values in each transition.
- Composition yields the conjunction of such constraints since in synchronous systems, the synchronized transitions occur simultaneously.

In this paper, we present a compositional semantics of synchronous systems that captures both safety and progress properties of such systems. The semantics is given in terms of the model of fair synchronous transitions systems (FSTS), which is based on the αSTS model [KP96]. The αSTS model has been used as a common semantic domain for both SIGNAL programs and the C-code generated by compiling them, for proving the correctness of the translation (compilation) [PSS98]. It has also been used in [BGA97] as a semantics which is "...fully general to capture the essence of the synchronous paradigm."

The FSTS model presented here extends αSTS by introducing operations for parallel composition, for the restriction of variables, and by addressing fairness. It can be used to answer questions such as:

- What is the valid set of runs corresponding to a given synchronous specification?
- How can we characterize a set of fair computations corresponding to a given synchronous specification?
- How can linear time temporal logic be adapted to reasoning about fair synchronous specifications?
- What is a sound compositional proof system for proving temporal safety and liveness properties of synchronous specifications?

The key characteristics of the FSTS model that make it suitable for capturing the semantics of synchronous systems are:

- Signals are modeled by variables whose value might be undefined in a state indicating the absence of the signal.
- The absence of a signal is indicated by having the values of variables range over a lifted domain that contains an undefined $\perp$ value in addition to defined data values.
- The variables of a transition system module are partitioned into
  - Synchronization variables that are used for interaction through input and output with other modules.

  - Controlled variables which are entirely controlled by the system consisting of variables that are externally visible and those that are local.

- Transitions can be taken on the basis of whether a signal is present or absent in a state.

- The crucial compositionality constraints on a FSTS module are :

  - A module transition cannot force the synchronization variables to take on defined values. It should always be possible for the system to enter a communication deadlock where the synchronization variables all remain undefined. This is because the module and its environment need to cooperate in order to synchronize on a defined value for a synchronization variable, and it is always possible for the environment to not cooperate with the module.
  - A module transition is invariant with respect to the part of the state that is unobservable by it. The behavior of the module is only affected by another module through the values (defined or undefined) of the synchronization variables that they share.
  - If any signal that is controlled by the module is continuously disabled, i.e., undefined, it must be because its definedness depends on the definedness of some subset of synchronization variables. This restriction is captured by means of a justice (weak fairness) condition that ensures that a variable that is continuously enabled to take on a defined value even when the synchronization variables are deadlocked, does eventually do so. The justice condition requires controlled variables to not deadlock, i.e., remain undefined, unless their definedness depends on the values of synchronization variables.

The FSTS model is designed to be simple and general. It extends the classical notion of transition systems with *signals* that can be present or absent in a given state, communication as *synchronization* by means of signals, *stuttering* as the absence of signals, and local progress through weak fairness constraints. The fairness condition ensures that any communication deadlock in a module can only occur through the need for external synchronization. Except for the fairness condition, the presence or absence of a signal is treated in a symmetrical manner as is the case in synchronous languages. The use of weak fairness constraints ensures that a module can satisfy these constraints without the cooperation of the environment, i.e., the module is *receptive* [AL95].

The FSTS model, the compositionality proofs, the extended linear temporal logic, and the accompanying soundness proofs have all been formally verified using PVS [Ow95]. The PVS verification pointed out a number of gaps in our earlier formalization and led to sharper definitions of the basic concepts, and

elegant and rigorous proofs.[1]

The paper is organized as follows. In Section 2 we introduce the FSTS computational model. In Section 3 we define two important operations on FSTS modules, *parallel composition* and *restriction*, and motivate the definitions by an intuitive example of progress (liveness). In Section 4 we present a formal method for proving temporal properties of synchronous systems by introducing an appropriate logic for expressing these properties and a system of deductive proof rules. We demonstrate the use of these rules by formalizing the intuitive arguments used in the example of Section 3. In Section 5 we present the FSTS semantics of the most expressive synchronous language – SIGNAL. (The FSTS semantics of ESTEREL and LUSTRE can be obtained in a similar way.)

2 Fair Synchronous Transition Systems

In this section, we introduce the notion of *fair synchronous transition systems*.

The Computational Model

We assume a vocabulary $\mathcal{V}$ which is a set of typed variables. Variables which are intended to represent signals are identified as *volatile*, and their domain contains a distinguished element $\perp$ used to denote the absence of the respective signal. In the translation of synchronous programs to FSTS specifications (see Section 5), we shall also use *persistent* variables to simulate the "memorization" operators of the synchronous languages (e.g., **current** in LUSTRE, "$" in SIGNAL).

Some of the types we consider are the *booleans* with domain $\mathcal{B} = \{\mathrm{T}, \mathrm{F}\}$, the type of *integers* whose domain Z consists of all the integers, the type of *pure signals* with domain $\mathrm{S}_\perp = \{\mathrm{T}, \perp\}$, the type of *extended booleans* with domain $\mathcal{B}_\perp = \{\mathrm{T}, \mathrm{F}, \perp\}$, and the type of *extended integers* with domain $\mathrm{Z}_\perp = Z \cup \{\perp\}$.

We define a *state* s to be a type-consistent interpretation of $\mathcal{V}$, assigning to each variable $u \in \mathcal{V}$ a value $s[u]$ over its domain. We denote by Σ the set of all states. For a subset of variables $V \subseteq \mathcal{V}$, we define a *V-state* to be a type-consistent interpretation of V.

Following [MP91] and [KP96], we define a *fair synchronous transition system* (FSTS) to be a system

$$\Phi\colon \quad \langle V, \Theta, \rho, E, S \rangle,$$

consisting of the following components:

- V : A finite set of typed *system variables.*
- Θ : The *initial condition*. A satisfiable assertion characterizing the initial states.

[1] The PVS formalization and proofs can be obtained from the URL `www.csl.sri.com/~singermn/fsts/`.

- ρ : A *transition relation*. This is an assertion $\rho(V, V')$, which relates a state $s \in \Sigma$ to its possible successors $s' \in \Sigma$ by referring to both unprimed and primed versions of the system variables. An unprimed version of a system variable refers to its value in s, while a primed version of the same variable refers to its value in s'. For example, the assertion $x' = x + 1$ states that the value of x in s' is greater by 1 than its value in s. If $\rho(s[V], s'[V]) = \mathtt{T}$, we say that state s' is a *ρ-successor* of state s.

  Remark: As implied by the notation $\rho(V, V')$, ρ can only refer to the variables of Φ, and therefore cannot distinguish between two possible Φ-successors that agree on the values of all the variables of Φ. That is, for all $s, s_1, s_2 \in \Sigma$,

$$s_1|V = s_2|V \rightarrow (\rho(s, s_1) \Leftrightarrow \rho(s, s_2)) .$$

- $E \subseteq V$: The set of *externally observable variables*. These are the variables that can be observed outside the module. We refer to $L = V - E$ as the *local variables*. Local variables cannot be observed outside the module.

- $S \subseteq E$: The set of *synchronization variables*. These are the signal variables on which the module may (and needs to) synchronize with its environment. We refer to $C = V - S$ as the *controllable variables*. These are the variables on whose values the module has full control.

For states $s, s' \in \Sigma$ and assertion $\varphi(V, V')$, we say that φ holds over the *joint interpretation* $\langle s, s' \rangle$, denoted

$$\langle s, s' \rangle \models \varphi,$$

if $\varphi(V, V')$ evaluates to T over the interpretation which interprets every $x \in V$ as $s[x]$ and every x' as $s'[x]$.

For an FSTS Φ, a state s' is called a *Φ-successor* of the state s if $\langle s, s' \rangle \models \rho$.

Definition 1 *An* FSTS Φ *is called* realizable *if every state $s \in \Sigma$ has a Φ-successor in which all the synchronization variables assume the value $\perp$. This requirement expresses the possibility that the environment might not be ready to co-operate with the module Φ in the current state.*

From now on, we restrict our attention to realizable FSTS specifications.

Computations of an FSTS

Let $\Phi = \langle V, \Theta, \rho, E, S \rangle$ be an FSTS. A *computation* of Φ is an infinite sequence

$$\sigma: \quad s_0, s_1, s_2, \ldots ,$$

where, for each $i = 0, 1, \ldots$, $s_i \in \Sigma$, and σ satisfies the following requirements:

- *Initiation*: s_0 is initial, i.e., $s_0 \models \Theta$.
- *Consecution*: For each $j = 0, 1, \ldots$, s_{j+1} is a Φ-successor of s_j.
- *Justice* (weak fairness): We say that a signal variable $x \in C$ is *enabled with respect to* S at state s_j of σ if there exists a Φ-successor s of s_j, such that $s[x] \neq \perp \wedge s[S] = \{\perp\}$.

  The *justice* requirement is that for every signal variable $x \in C$, it is not the case that x is enabled w.r.t. S in all but a finite number of states along σ, while $s_j[x] \neq \perp$, for only finitely many states s_j in the computation.

Remark: In the sequel, we shall sometimes use the term "enabled w.r.t. Φ" instead of "enabled w.r.t. S_Φ".

The requirement of justice with respect to a controllable signal variable x demands that if x is continuously enabled to assume a non-$\perp$ value, from a certain point on, without the need to synchronize with the environment, it will eventually assume such a value. The fact that a variable x is enabled to become non-$\perp$ without 'external assistance' is evident from the existence of a Φ-successor s such that $s[x] \neq \perp \wedge s[S] = \{\perp\}$.

A *run* of a system Φ is a finite or an infinite sequence of states satisfying the requirements of initiation and consecution, but not necessarily the requirement of justice.

An FSTS is called *viable* if every finite run can be extended into a computation. From now on, we restrict our attention to viable FSTS specifications.

For the case that Φ is a finite-state system, i.e., all system variables range over finite domains, there exist (reasonably) efficient symbolic model-checking algorithms for testing whether Φ is viable.

A state s' is a *stuttering variant* of a state s if all volatile (i.e., signal) variables are undefined in s' and s' agrees with s on all persistent variables. A state sequence $\widehat{\sigma}$ is said to be a *stuttering variant* of the state sequence $\sigma : s_0, s_1, \ldots$, if $\widehat{\sigma}$ can be obtained from σ by repeated transformations where a state s in σ is replaced with s, s' in $\widehat{\sigma}$, or a pair of adjacent states s, s' in σ is replaced with s in $\widehat{\sigma}$ where s' is a stuttering variant of s. A set of state sequences S is called *closed under stuttering* if, for every state sequence σ, $\sigma \in S$ iff all stuttering variants of σ belong to S. An FSTS Φ is called *stuttering robust* if the set of Φ-computations is closed under stuttering.

3 Operations on FSTS Modules

There are two important operations on FSTS modules: *parallel composition* and *restriction*.

3.1 Parallel Composition

Let Φ_1: $\langle V_1, \Theta_1, \rho_1, E_1, S_1 \rangle$ and Φ_2: $\langle V_2, \Theta_2, \rho_2, E_2, S_2 \rangle$ be two FSTS modules. These systems are called *syntactically compatible* if they satisfy

$$C_1 \cap V_2 = V_1 \cap C_2 = \emptyset \text{ (or equivalently } V_1 \cap V_2 = S_1 \cap S_2).$$

That is, only synchronization variables can be common to both systems. We define the *parallel composition* of syntactically compatible Φ_1 and Φ_2, denoted $\Phi = \Phi_1 \parallel \Phi_2$, to be the FSTS Φ: $\langle V, \Theta, \rho, E, S \rangle$, where

$$\begin{array}{lcl} V & = & V_1 \cup V_2 \\ \Theta & = & \Theta_1 \wedge \Theta_2 \\ \rho & = & \rho_1 \wedge \rho_2 \\ E & = & E_1 \cup E_2 \\ S & = & S_1 \cup S_2 \end{array}$$

To indicate the relation between computations of a composed system and computations of its constituents, we first prove the following lemma:

Lemma 1 *Let Φ_1: $\langle V_1, \Theta_1, \rho_1, E_1, S_1 \rangle$ and Φ_2: $\langle V_2, \Theta_2, \rho_2, E_2, S_2 \rangle$ be two compatible* FSTS *modules, x be a controlled signal variable of Φ_1, and σ: $s_0, s_1, s_2, \ldots$ be a sequence of states. For every $j \geq 0$:*

$$x \text{ is enabled in } s_j \text{ w.r.t. } \Phi_1 \iff x \text{ is enabled in } s_j \text{ w.r.t. } \Phi_1 \parallel \Phi_2.$$

Proof: ($\Longrightarrow$) First, note that since x is a controlled signal variable of Φ_1, it is also a controlled signal variable of $\Phi_1 \parallel \Phi_2$. Now, assume that s is a Φ_1-successor of s_j, s.t. $s[x] \neq \bot$ and $s[S_1] = \bot$. By the realizability requirement applied to Φ_2 (see Definition 1), there exists a state s', s.t. $\rho_2(s, s')$ and $s'[S_2] = \bot$. Let s'' be the state that valuates each variable of Φ_1 as s and all other variables as s'. Since Φ_1 and Φ_2 are syntactically compatible, we have $s''[x] \neq \bot$ and $s''[S_1 \cup S_2] = \bot$. Hence, by recalling that ρ_1 and ρ_2 can only refer to variables in V_1 and V_2, respectively, we have that $\rho_1(s, s'')$ and $\rho_2(s, s'')$; i.e., s'' is a $\Phi_1 \parallel \Phi_2$-successor of s_j, and we are done.

The argument in the other direction is similar. ∎

We can now prove the following theorem:

Theorem 2 *Let Φ_1: $\langle V_1, \Theta_1, \rho_1, E_1, S_1 \rangle$ and Φ_2: $\langle V_2, \Theta_2, \rho_2, E_2, S_2 \rangle$ be two compatible* FSTS *modules, and σ: $s_0, s_1, s_2, \ldots$ be a sequence of states.*

σ is a computation of $\Phi_1 \parallel \Phi_2$ $\iff$ σ is both a computation of Φ_1 and of Φ_2.

Proof: ($\Longrightarrow$) Due to symmetry, it suffices to prove that σ is a computation of Φ_1. The fact that σ satisfies the initiation and consecution requirements for Φ_1 follows easily from the definition of $\Phi_1 \parallel \Phi_2$. What remains is to show that σ is fair (just) w.r.t. Φ_1. Let x be a controlled variable of Φ_1 which is continuously

enabled w.r.t. Φ_1. We have to prove that $x \neq \bot$ infinitely often. By Lemma 1, we conclude that x is continuously enabled w.r.t. $\Phi_1 \parallel \Phi_2$. So, by the assumption that σ is a computation of $\Phi_1 \parallel \Phi_2$, and is therefore just w.r.t. $\Phi_1 \parallel \Phi_2$, we have $x \neq \bot$ infinitely often.

($\Longleftarrow$) Again, it is easy to see that σ satisfies the the initiation and consecution requirements for $\Phi_1 \parallel \Phi_2$. To prove that σ is fair w.r.t. $\Phi_1 \parallel \Phi_2$, let x be a controlled variable of $\Phi_1 \parallel \Phi_2$ which is continuously enabled w.r.t. $\Phi_1 \parallel \Phi_2$. By the definition of $\Phi_1 \parallel \Phi_2$, $x \in (V_1 \setminus S_1) \cup (V_2 \setminus S_2)$. Without loss of generality, assume $x \in (V_1 \setminus S_1)$, i.e., x is a controlled variable of Φ_1. Using Lemma 1, we see that x is continuously enabled w.r.t. Φ_1, so that by the fact that σ is fair w.r.t. to Φ_1, we have $x \neq \bot$ infinitely often. ∎

3.2 Restriction

In the operation of restriction, we identify a set of synchronization variables and close off the system for external synchronization on these variables.

Let $W \subseteq S$ be a set of synchronization variables of the FSTS Φ: $\langle V, \Theta, \rho, E, S \rangle$. We define the *$W$-restriction* of Φ, denoted [**own** W: Φ], to be the FSTS $\widetilde{\Phi}$: $\langle \widetilde{V}, \widetilde{\Theta}, \widetilde{\rho}, \widetilde{E}, \widetilde{S} \rangle$, where

$$\begin{array}{lcl} \widetilde{V} & = & V \\ \widetilde{\Theta} & = & \Theta \\ \widetilde{\rho} & = & \rho \\ \widetilde{E} & = & E \\ \widetilde{S} & = & S \setminus W. \end{array}$$

Thus the effect of W-restricting the system Φ amounts to moving the variables in W from S to C. This movement may have a considerable effect on the computations of the system.

Example 1 Consider the FSTS Φ, where

$$\begin{array}{rcl} V = E = S & : & \{x : \{\mathtt{T}, \bot\}\} \\ \Theta & : & x = \mathtt{T} \\ \rho & : & x' = \mathtt{T} \ \vee \ x' = \bot. \end{array}$$

Let $\widetilde{\Phi}$: [**own** x. Φ] denote the x-restriction of Φ. The sequence below, in which next to each state appears the set of all possible successors,

$$\begin{array}{lll} \sigma: & s_0\colon \langle x\colon \mathtt{T} \rangle, & \{\langle x\colon \bot \rangle, \langle x\colon \mathtt{T} \rangle\} \\ & s_1\colon \langle x\colon \bot \rangle, & \{\langle x\colon \bot \rangle, \langle x\colon \mathtt{T} \rangle\} \\ & s_2\colon \langle x\colon \bot \rangle, & \ldots \end{array}$$

is a computation of Φ but is not a computation of $\widetilde{\Phi}$. Note that x assumes a non-$\bot$ value only at s_0. The sequence σ is a computation of Φ since x is not in

C ($x \in S$), and therefore is not enabled w.r.t. S in any state of σ. On the other hand, x is no longer a synchronization variable in $\widetilde{\Phi}$, since it belongs to $\widetilde{C}$. The sequence σ is not a computation of $\widetilde{\Phi}$ since it is unfair to x. This is because now, x is enabled w.r.t. $S_{\widetilde{\Phi}}$ in every state of σ, but $s[x] \neq \bot$ only in s_0. From this we can deduce that all computations of $\widetilde{\Phi}$ contain infinitely many states in which $x \neq \bot$, but this is not necessarily the case for computations of Φ.

The following lemma (whose proof is left for the final version) describes the relation between computations of unrestricted and restricted systems.

Lemma 3 *The infinite sequence*

$$\sigma: \quad s_0, s_1, s_2, \ldots$$

is a computation of [**own** $W.\Phi$] *iff*

1. *σ is a computation of Φ, and*
2. *σ satisfies the justice requirement w.r.t. $S \setminus W$.*

Thus, the computations of [**own** $W.\Phi$] can be obtained from those of Φ.

Example 2 Consider the FSTS Φ_1 defined by

$$\begin{array}{rcl} V_1 = E_1 & : & \{x, y : \mathrm{Z}_\bot\} \\ S_1 & : & \{x\} \\ \Theta_1 & : & x = y = \bot \\ \rho_1 & : & (y' = 3) \wedge (x' = 4) \quad \vee \quad (y' = \bot) \wedge (x' \neq 4) \end{array}$$

and the FSTS Φ_2 defined by

$$\begin{array}{rcl} V_2 = E_2 = S_2 & : & x : \mathrm{Z}_\bot \\ \Theta_2 & : & x = \bot \\ \rho_2 & : & (x' = 4) \vee (x' = 5) \vee (x' = \bot). \end{array}$$

Both of these FSTS modules have computations satisfying $\Box \Diamond (y = 3)$. There exists, however, a computation of both Φ_1 and Φ_2 which by Theorem 2 is therefore also a computation of $\Phi_1 \parallel \Phi_2$, which does not satisfy $\Box \Diamond (y = 3)$. This computation is

$$\begin{array}{rll} \sigma: & s_0: \langle \overset{x}{\bot}, \overset{y}{\bot} \rangle, & \{\langle \bot, \bot \rangle, \langle 4, 3 \rangle, \langle 5, \bot \rangle\} \\ & s_1: \langle \bot, \bot \rangle, & \{\langle \bot, \bot \rangle, \langle 4, 3 \rangle, \langle 5, \bot \rangle\} \\ & s_2: \langle \bot, \bot \rangle, & \ldots \end{array}$$

In σ, the variable y does not get the value 3 even once. This is fair, since y is not enabled w.r.t. $S_{\Phi_1 \| \Phi_2}$ in any state of σ. This is because $y' \neq \bot \wedge x' = \bot$ is

false in every state of σ. Now, suppose we close off Φ with respect to x, to get the FSTS module

$$\widetilde{\Phi}: \quad [\mathbf{own}\ x.[\Phi_1 \parallel \Phi_2]].$$

In $\widetilde{\Phi}$, x is no longer a synchronization variable, and therefore y is continuously enabled w.r.t. $S_{\widetilde{\Phi}}$ in σ (in every state, in fact). However, it is obviously not the case that $y \neq \perp$ is satisfied infinitely often in σ, and hence σ is not a computation of $\widetilde{\Phi}$. In conclusion, we see that only the restriction of x guarantees $\Box \Diamond (y = 3)$.

4 Compositional Verification of FSTS Modules

In this section we show how to construct compositional verification of the temporal properties of FSTS's, concentrating on liveness properties.
First, we define an appropriate logic. Let ELTL be LTL extended with the unary predicate *ready*. An ELTL *model* is a pair $M = (L, \sigma)$, where σ is an infinite sequences of the form

$$\sigma: \quad s_0, s_1, s_2, \ldots, \text{ where } s_j \in \Sigma, \text{ for every } j \geq 0,$$

and $L : \Sigma \to 2^{\Sigma}$. ELTL formulas are interpreted as follows.

- For a state formula p, $(M, j) \models p \ \leftrightarrow\ s_j \models p$.
- $(M, j) \models \neg p \ \leftrightarrow\ (M, j) \not\models p$.
- $(M, j) \models p \vee q \ \leftrightarrow\ (M, j) \models p$ or $(M, j) \models q$.
- $(M, j) \models \bigcirc p \ \leftrightarrow\ (M, j + 1) \models p$.
- $(M, j) \models p \mathcal{U} q \ \leftrightarrow\ (M, k) \models q$ for some $k \geq j$ and $(M, i) \models p$ for all i, $j \leq i < k$.
- For a state formula p, $(M, j) \models \mathit{ready}(p) \ \leftrightarrow\ \exists s \in L(s_j)$ s.t. $s \models p$.

As usual, we use the abbreviations $\Diamond p$ for $\mathrm{T}\mathcal{U} p$ and $\Box p$ for $\neg \Diamond \neg p$.

We say that a model M *satisfies* an ELTL formula p, written $M \models p$, if $(M, 0) \models p$.

Notation: For an ELTL model $M = (L, \sigma)$, we denote by $\sigma(i)$, the $i + 1$-th element of σ.

For an FSTS Φ, the ELTL model $M = (L, \sigma)$ is called a Φ-*model*, if

1. σ is a computation of Φ, and

1.	$\Phi_1 \models \Diamond\Box \mathit{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n)$, where $\{z_1, \ldots, z_n\} \supseteq S_{\Phi_1} \cap S_{\Phi_2}$
2.	$\Phi_2 \models \Diamond\Box \mathit{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n)$
	$\Phi_1 \parallel \Phi_2 \models \Diamond\Box \mathit{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n)$

Figure 1: Rule READY.

2. For every $j = 0, 1, 2, \ldots$, $L(\sigma(j))$ is the set of all possible Φ-successors of $\sigma(j)$, i.e., $L(\sigma(j)) = \text{image}(\rho, \{\sigma(j)\})$.

We write $\Phi \models p$, and say that p *is valid over* Φ, if $M \models p$, for every Φ-*model* M.

Definition 2 *An* ELTL *formula is called* universal, *if it does not contain* ready *or if each occurrence of* ready *appears under an* odd *number of negations.*

We present our method by formalizing the intuitive arguments used in Example 2. In the proof, we introduce several deductive rules that we believe may be useful in typical liveness proofs. The soundness of these rules is proved in the sequel.

Let us begin by noting that

$$\Phi_1, \Phi_2 \models \Box \mathit{ready}(y = 3 \wedge x = 4), \tag{1}$$

can be verified independently for Φ_1 and for Φ_2. From (1) and the temporal tautology $q \rightarrow \Diamond q$, we can derive

$$\Phi_1, \Phi_2 \models \Diamond\Box \mathit{ready}(y = 3 \wedge x = 4). \tag{2}$$

Applying rule READY, presented in Fig. 1, to (2) yields

$$\Phi_1 \parallel \Phi_2 \models \Diamond\Box \mathit{ready}(y = 3 \wedge x = 4). \tag{3}$$

From the latter, we can easily derive

$$\Phi_1 \parallel \Phi_2 \models \Diamond\Box \mathit{ready}(y \neq \bot). \tag{4}$$

By applying rule OWN (Fig. 2) to (4), with $W = \{x\}$, we get

$$\underbrace{[\textbf{own}\ x.[\Phi_1 \parallel \Phi_2]]}_{\Phi} \models \Diamond\Box \mathit{ready}(y \neq \bot). \tag{5}$$

Now, since $S_\Phi = \emptyset$, we can use axiom CONT (Fig. 3), and (5) with $z = y$, to derive

$$\Phi \models \Box\Diamond(y \neq \bot). \tag{6}$$

$$\frac{\Phi \models p, \text{ where } p \in \text{ELTL}}{[\textbf{own } W.\Phi] \models p}$$

Figure 2: Rule OWN.

$$\Phi \models \Diamond \Box \mathit{ready}(z \neq \bot \wedge S_\Phi = \bot) \rightarrow \Box \Diamond (z \neq \bot), \text{ where } z \in C_\Phi$$

Figure 3: Axiom CONT.

It is not difficult to prove

$$\Phi_1 \models \Box(y = 3 \vee y = \bot). \tag{7}$$

By applying rule COMP (Fig. 4) to the latter, we get

$$\Phi_1 \parallel \Phi_2 \models \Box(y = 3 \vee y = \bot). \tag{8}$$

We now apply rule OWN (Fig. 2) to (8), to get

$$\Phi \models \Box(y = 3 \vee y = \bot). \tag{9}$$

The latter, together with (6) implies $\Phi \models \Box \Diamond (y = 3)$ which completes the proof.

Now, as promised, we prove the soundness of the deductive rules. We start with two lemmas that follow from Theorem 2, and characterize the relation between ELTL models of a composed system and those of its constituents.

Lemma 4 *Let Φ_1 and Φ_2 be two compatible fsts modules, $M_1 = (L_1, \sigma)$ be a Φ_1-model, and $M_1 = (L_2, \sigma)$ be a Φ_2-model. Then*

$(L_\cap, \sigma)$ *is a $\Phi_1 \parallel \Phi_2$ -model, where $L_\cap(s) = L_1(s) \cap L_2(s)$, for every $s \in \Sigma$.*

Lemma 5 *Let Φ_1 and Φ_2 be two compatible fsts modules, and $M = (L, \sigma)$ be a $\Phi_1 \parallel \Phi_2$-model. Then*

(L_{Φ_1}, σ) *is a Φ_1 -model, where $L_{\Phi_1}(s) = \mathit{image}(\rho_{\Phi_1}, \{s\})$, for every $s \in \Sigma$.*

Theorem 6 *The rule* READY *is sound. That is, let Φ_1: $\langle V_1, \Theta_1, \rho_1, E_1, S_1 \rangle$ and Φ_2: $\langle V_2, \Theta_2, \rho_2, E_2, S_2 \rangle$ be two compatible* FSTS*'s, and $\{z_1, \ldots, z_n\}$ be a set of*

$$\frac{\Phi_1 \models p, \text{ where } p \in \text{ELTL is universal}}{\Phi_1 \parallel \Phi_2 \models p}$$

Figure 4: Rule COMP.

variables s.t. $\{z_1, \ldots, z_n\} \supseteq S_1 \cap S_2$:

$$\begin{array}{l}(\Phi_1 \models \Diamond \Box \,\text{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n)) \wedge \\ (\Phi_2 \models \Diamond \Box \,\text{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n))) \;\rightarrow \\ (\Phi_1 \parallel \Phi_2 \models \Diamond \Box \,\text{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n)).\end{array}$$

Proof: [(sketch)] Let $M = (L, \sigma)$ be a $\Phi_1 \parallel \Phi_2$-model. Use Lemma 5 to construct a Φ_1-model, $M_1 = (L_{\Phi_1}, \sigma)$, and a Φ_2-model, $M_2 = (L_{\Phi_2}, \sigma)$. By the assumption,

$$\begin{array}{lll} M_1 & \models & \Diamond \Box \, \mathit{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n) \text{ and} \\ M_2 & \models & \Diamond \Box \, \mathit{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n). \end{array}$$

Hence, there exists an i_0 s.t. for all $j \geq i_0$,

$$\begin{array}{lll} (M_1, j) & \models & \mathit{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n) \text{ and} \\ (M_2, j) & \models & \mathit{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n). \end{array}$$

Note that

$$\{z_1, \ldots, z_n\} \supseteq S_1 \cap S_2 \supseteq V_1 \cap V_2,$$

so that using the realizability requirement (see Def. 1), it is not too difficult to prove that for every $j \geq i_0$:

$$(((L_{\Phi_1} \cap L_{\Phi_2}, \sigma), j) \models \mathit{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n).$$

That is

$$M \models \Diamond \Box \, \mathit{ready}(z_1 = c_1 \wedge \ldots \wedge z_n = c_n).$$

■

Theorem 7 *The rule* OWN *(Fig. 2) is sound. That is, let* Φ: $\langle V, \Theta, \rho, E, S \rangle$ *be an* FSTS, $W \subseteq S$ *be a set of variables, and* p *be an* ELTL *formula. Then*

$$(\Phi \models p) \;\rightarrow ([\textbf{own}\ W.\Phi] \models p)\,.$$

Proof: Let $M_{\text{own}} = (L, \sigma)$ be a [**own** $W.\Phi$]-model. First, note that σ is a computation of [**own** $W.\Phi$], so by Lemma 3 it is also a computation of Φ. Next, recall that for every $j \geq 0$, $L(\sigma(j))$ is the set of all possible [**own** $W.\Phi$]-successors of $\sigma(j)$; but since Φ and [**own** $W.\Phi$] have the same transition relation, $L(\sigma(j))$ is also the set of all possible Φ-successors of $\sigma(j)$. Thus, M_{own} is a Φ-model, and by the assumption $M_{\text{own}} \models p$. ∎

Theorem 8 *The axiom* CONT *(Fig. 3) is valid. That is, let* Φ: $\langle V, \Theta, \rho, E, S\rangle$ *be an* FSTS*, and* z *be a controlled variable of* Φ*. Then*

$$\Phi \models \Diamond\Box \,\text{ready}(z \neq \bot \wedge S = \bot) \rightarrow \Box\Diamond(z \neq \bot).$$

Proof: (sketch) Suppose that $M = (L, \sigma)$ is an Φ-model, and $z \in C_\Phi$. σ is a computation of Φ, and is therefore fair w.r.t. z. The observant reader would notice that

$$\Diamond\Box \, \mathit{ready}(z \neq \bot \wedge S = \bot) \rightarrow \Box\Diamond(z \neq \bot)$$

is simply a reformulation of the justice requirement w.r.t. z. From this it is not difficult to complete the proof. ∎

To prove the soundness of the COMP, we need some preparation.

Definition 3 *An* ELTL *formula is called* existential*, if it does not contain* ready *or if each occurrence of* ready *appears under an* even *number of negations.*

Lemma 9 *Let* $M_1 = (\sigma, L_1)$ *and* $M_2 = (\sigma, L_2)$ *be two* ELTL *models, s.t.* $L_2(\sigma(j)) \subseteq L_1(\sigma(j))$*, for each* $j = 0, 1, 2, \ldots$*. Then, for every* ELTL *formula* p*, and every* $i = 0, 1, 2, \ldots$*:*

1. $((M_1, i) \models p \wedge (p \textit{ is universal}) \rightarrow (M_2, i) \models p)$*, and*
2. $((M_2, i) \models p \wedge (p \textit{ is existential}) \rightarrow (M_1, i) \models p)$.

Proof: [(sketch)] The proof is carried out by mutual induction on the structure of p. Let us only mention that for the $p = \neg(q)$ case, observe that if $\neg(q)$ is universal then q is existential, and if $\neg(q)$ is existential then q is universal. Hence, (1) follows from the induction hypothesis for (2), and vice-versa. ∎

Theorem 10 *The rule* COMP *(Fig. 4) is sound. That is, let* Φ_1 *and* Φ_2 *be two compatible fsts modules, and* p *be a universal* ELTL *formula. Then*

$$(\Phi \models p) \;\rightarrow\; (\Phi_1 \parallel \Phi_2 \models p).$$

Proof: Suppose $M_\parallel = (L, \sigma)$ is a $\Phi_1 \parallel \Phi_2$-model. By Lemma 5, the ELTL model $M_1 = (L_\Phi, \sigma)$, where $L_{\Phi_1}(s) = \text{image}(\rho_{\Phi_1}, \{s\})$, for every $s \in \Sigma$, is a Φ_1-model. So, by the assumption, $M_1 \models p$. Now, for every state $s \in \Sigma$:

$$L(s) \;=\; \text{image}(\rho_{\Phi_1 \parallel \Phi_2}, \{s\}) \;\subseteq\; \text{image}(\rho_{\Phi_1}, \{s\}) \;=\; L_{\Phi_1}(s).$$

Hence, recall that p is universal, so that by using Lemma 9 (with $M_2 = M_{\|}$ and $i = 0$), we can conclude $M_{\|} \models p$. ∎

We conclude with a practical remark. Verification of FSTS specification can be done by using existing symbolic model-checking algorithms. Computing the L-sets comes at no extra cost, since the predicate *ready*(p) is equivalent to the CTL formula EXp which every model checker knows how to compute very efficiently.

5 The FSTS semantics of SIGNAL

As mentioned in the introduction, the FSTS model is a significant extension of the previous, more basic, αSTS model [KP96] obtained by introducing operations (parallel composition and restriction) and by addressing fairness. The translation from SIGNAL programs to corresponding FSTS specifications, however, is not affected by these extensions, and can be carried out exactly as with αSTS simply by taking the input/output variables as the *externally observable* variables and also as the *synchronization* variables. Nevertheless, to make the paper more self-contained, we present below the translation given by Kesten and Pnueli [KP96].

For a variable v, *clocked*(v) denotes the assertion:

$$clocked(v) : v \neq \perp$$

In the following, we describe how to construct an FSTS Φ_P corresponding to a given SIGNAL program P.

System Variables

The system variables of Φ are given by $V = U \cup X$, where U are the SIGNAL variables explicitly declared and manipulated in P, and X is a set of auxiliary variables. An auxiliary variable by the name of $x.v$ is included in X for each expression of the form v \$ appearing in P. For simplicity, we assume that the \$ operator is only applied to variables and not to more general expressions. The value of $x.v$ is intended to represent the value of v at the previous instance (present excluded) that v was different from $\perp$.

Externally observable and synchronization variables

The externally observable variables E and also the synchronization variables S are those explicitly declared in P as input/output variables.

Initial Condition

The initial condition for Φ is given by

$$\Theta: \quad \bigwedge_{u \in U} u = \bot \quad \wedge \quad \bigwedge_{x.v \in U} x.v = \bot$$

As will result from our FSTS translation of SIGNAL programs, they are all stuttering robust. Consequently, we can simplify things by assuming that the first state in each run of the system is a stuttering state.

Transition Relation

The transition relation ρ will be a conjunction of assertions, where each SIGNAL statement gives rise to a conjunct in ρ.

We list the statements of SIGNAL and, for each statement S, we present the conjunct contributed to ρ by S.

Basic Instructions

- Consider the SIGNAL statement $y := f(v_1, \ldots, v_n)$, where f is a state-function. Its contribution to ρ is given by:

$$\begin{array}{ll} & \mathit{clocked}(y') \equiv \mathit{clocked}(v'_1) \equiv \ldots \equiv \mathit{clocked}(v'_n) \\ \wedge & (\mathit{clocked}(y') \rightarrow y' = f(v'_1, \ldots, v'_n)) \end{array}$$

  This formula requires that the signals $y, v_1, \ldots, v_n$ are present at precisely the same time instants, and that at these instants $y = f(v_1, \ldots, v_n)$.

- The contribution of the statement

$$y := v\$ \ \textbf{init} \ v_0$$

  is given by:

$$\begin{array}{lll} x'.v & = & \textbf{if } \mathit{clocked}(v') \textbf{ then } v' \textbf{ else } x.v \\ \wedge \quad y' & = & \left(\begin{array}{lll} \textbf{if} & \neg \mathit{clocked}(v') & \textbf{then} \quad \bot \\ \textbf{else if} & x.v = \bot & \textbf{then} \quad v_0 \\ \textbf{else} & x.v & \end{array} \right) \end{array}$$

  The first conjunct of this formula defines the new value of $x.v$. If the new value of v is different from $\bot$, then the new value of $x.v$ is the new value of v. Otherwise, $x.v$ retains its current value.

  The second conjunct of the formula defines the new value of y by considering three cases. The first case requires that $y' = \bot$ whenever $v' = \bot$. This together with the other two cases implies that the clocks of v and

y are identical. The second considered case is the first position at which $v' \neq \bot$. Observe that the fact that we are at the *else* clause of the test $\neg clocked(v')$ implies that $v' \neq \bot$, and that $x.v = \bot$ implies that there was no previous position at which $v \neq \bot$. In this case, we take the new value of y (y') to be v_0. The last case considers subsequent positions at which $v' \neq \bot$. At all of these positions, y' is taken to be the value of $x.v$, i.e., the value of v as it were at the previous ($v' \neq \bot$)-position and as memorized in $x.v$.

- The contribution of the statement

$$y := v \textbf{ when } b$$

is given by:

$$y' = \textbf{ if } (b' = \mathrm{T}) \textbf{ then } v' \textbf{ else } \bot.$$

- The contribution of the statement

$$y := u \textbf{ default } v$$

is given by:

$$y' = \textbf{ if } clocked(u') \textbf{ then } u' \textbf{ else } v'.$$

Shorthand Instructions

- The contribution of the statement

$$\varphi(\textbf{synchro } v, y),$$

which states that v and y have the same clock, is given by:

$$clocked(v') \equiv clocked(y')$$

- The contribution of the statement

$$y := \textbf{when}(v),$$

which is an abbreviation for: $y := v$ when v, for a boolean variable v, is given by:

$$y' = \textbf{ if } (v' = \mathrm{T}) \textbf{ then } \mathrm{T} \textbf{ else } \bot.$$

- The contribution of the statement

$$y := \textbf{ event}(v),$$

which defines y to be a pure signal representing the clock of v, is given by:

$$y' = \textbf{ if } clocked(v') \textbf{ then } \mathrm{T} \textbf{ else } \bot.$$

GUARD_COUNT {**input event** *fill*, **output boolean** *empty*}

=

synchro (**when**$(zn = 0)$), *fill*
| $n :=$ (10 **when** *fill*) **default** $(zn - 1)$
| $zn :=$ n \$ **init** 0
| *empty* := **when**$(n = 0)$ **default** (**not** *fill*)

Figure 5: A sample SIGNAL program.

Example

In Fig. 5, we present a SIGNAL program example, taken from [BGJ91]. We simplified the program to make it more self-contained. This small program models a system with a replenishable resource, for example, a water reservoir. The input event *fill* signals that that the reservoir is filled to the top. The local integer variable n measures the current water level. At each *fill* signal, the level is set to 10 (assumed maximal capacity). Then the level gradually decreases until it reaches 0. The output signal *empty* will register T when the water level drops to 0, and will register F when the reservoir is next filled.

The program as an FSTS

The FSTS translation of the SIGNAL program of Fig. 5 is defined as follows:

The system variables are given by:

$$V: \quad \{\underbrace{\mathit{fill}, \mathit{empty}, zn, n}_{U}, \underbrace{x.n}_{X}\}.$$

The externally observable and synchronization variables are given by:

$$E = S: \quad \{\mathit{fill}, \mathit{empty}\}.$$

The initial condition is given by:

$$\Theta: \quad \mathit{fill} = \mathit{empty} = zn = n = x.n = \bot.$$

The transition relation ρ is given by:

$$
\begin{array}{lrcl}
 & (zn' = 0) & \equiv & clocked(fill') \\
\wedge & n' & = & \left(\begin{array}{llll} \textbf{if} & fill' = \mathtt{T} & \textbf{then} & 10 \\ \textbf{else if} & clocked(zn') & \textbf{then} & zn' - 1 \\ \textbf{else} & \bot & & \end{array} \right) \\
\wedge & x.n' & = & \textbf{if } clocked(n') \textbf{ then } n' \textbf{ else } x.n \\
\wedge & zn' & = & \left(\begin{array}{llll} \textbf{if} & \neg clocked(n') & \textbf{then} & \bot \\ \textbf{else if} & x.n = \bot & \textbf{then} & 0 \\ \textbf{else} & x.n & & \end{array} \right) \\
\wedge & empty' & = & \left(\begin{array}{llll} \textbf{if} & n' = 0 & \textbf{then} & \mathtt{T} \\ \textbf{else if} & fill' = \mathtt{T} & \textbf{then} & \mathtt{F} \\ \textbf{else} & \bot & & \end{array} \right)
\end{array}
$$

6 Conclusions and Future Work

We have presented FSTS, a compositional semantics of synchronous systems that captures both safety and progress properties. We have motivated the fairness requirement and the operations of parallel composition and of restriction of variables by means of intuitive examples.

We have then introduced an extended version of linear temporal logic (ELTL), in which it is convenient to express safety and liveness properties of synchronous specifications, and have presented (and demonstrated) a sound compositional proof system for it.

We have concluded by specifying how to translate programs written in an expressive representative of the synchronous school, namely SIGNAL, to FSTS.

Directions in future work which we intend to pursue are

- Specifying in detail the FSTS semantics of LUSTRE, ESTEREL and STATECHARTS.
- Apply the deductive proof system developed here together with existing symbolic model-checking algorithms to the verification of FSTS specifications that result from actual synchronous programs.

References

[AL95] M. Abadi and L. Lamport. Conjoining Specifications. *TOPLAS*, 17(3), pages 507–534, 1995.

[BGA97] A. Benveniste, P. Le Guernic, and P. Aubry. Compositionality in dataflow synchronous languages: specification & code generation. *Proceedings of COMPOS'97.*

[BGJ91] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with event and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16, pages 103–149, 1991.

[BG92] G. Berry and G. Gonthier. The ESTEREL Synchronous Programming Language: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a Declarative Language for Programming Synchronous Systems. *POPL'87*, ACM Press, pages 178–188, 1987.

[H93] N. Halbwachs. *Synchronous Programming of Reactive Systems.* Kluwer, Dordrecht, The Netherlands, 1993.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, pages 231–274, 1987.

[KP96] Y. Kesten and A. Pnueli. An αSTS-based common semantics for SIGNAL and STATECHARTs, March 1996. Sacres Manuscript.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, New York, 1991.

[Ow95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. IEEE trans. on software eng., 21(2), pages 107–125, 1995.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. *TACAS'98*, LNCS, 1998.

Reconfiguration and Transient Recovery in State Machine Architectures*

From *Fault Tolerant Computing Symposium 26*, pp. 6–15 Sendai, Japan, June 1996.

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

We consider an architecture for ultra-dependable operation based on synchronized state machine replication, extended to provide transient recovery and reconfiguration in the presence of arbitrary faults.

The architecture allows processors suspected of being faulty to be placed on "probation." Processors in this status cannot disrupt other processors, but those that are nonfaulty or recovering from transient faults are able to remain synchronized with the other processors and with each other, can participate in interactively consistent exchange of data (i.e., Byzantine agreement), and can restore damaged state data by loading majority-voted copies from other processors. The processors that are not on probation are able to coordinate membership of their group and to take processors on and off probation. These properties are achieved even if all the processors on probation and some of the others exhibit Byzantine faults, provided a majority of all processors are nonfaulty.

Key elements of the architecture are modified treatments for the problems of interactive consistency, clock synchronization, and group membership. Classical algorithms for these problems that tolerate t Byzantine faults among n processors are extended to tolerate $t+p$ faults among $n+p$ processors, partitioned into n "core members" and p "probationers," provided no more than t faults occur among the core members.

1 Introduction

There are two basic approaches to the design of systems that can tolerate multiple faults: the system must either have sufficient redundancy that it can mask multiple simultaneous faults, or it must diagnose faulty components and reconfigure to exclude them from the system before additional faults arrive.[1] The advantage of diagnosis and reconfiguration over multiple-fault masking is that it requires far less redundancy to tolerate the same number of faults. For example, a multiple-fault masking system requires $2t+1$ components to tolerate t simultaneous faults in those elements of its architecture that use majority voting, and $3t+1$ components in those elements that require Byzantine fault-tolerant clock synchronization or interactive consistency (under "Oral Messages" assumptions). A system based on aggressive reconfiguration[2] could tolerate the same number of faults in the corresponding elements of its architecture with only $t+2$, and $t+3$ components, respectively. Even one that (as here) focuses only on increasing the resilience of clock synchronization and interactive consistency can reduce the number of components required from $3t+1$ to $2t+1$ for $t>1$.

One serious disadvantage of the reconfiguration approach, however, is that it assumes that faults arrive sequentially and that a faulty component can be diagnosed and excluded before the next fault arrives: the approach will fail if several faults arrive in close proximity, or if diagnosis is inaccurate or incomplete. Unfortunately, it is provably impossible to achieve both accurate (no nonfaulty components are identified as faulty) and complete (all faulty components are identified) diagnosis in the presence of Byzantine faults [23].

Another disadvantage of simple reconfiguration is that it provides poor treatment for transient faults—which experience shows to be far more common than permanent faults. A transiently faulty component is one that malfunctions for a while and then spontaneously recovers. However, the processor in such a re-

*This work was partially supported by NASA Langley Research Center under contract NAS1-18969 and by the Air Force Office of Scientific research under contract F49620-95-C0044.

[1] *Diagnosis* is used here to refer simply to the detection and identification of faulty components; it does not imply identification of the type of fault that afflicts the component. A *component* is used here to refer to an active computing element that includes a processor and local memory.

[2] That is, one that permanently excludes components diagnosed as faulty from all participation (including voting on outputs).

covered component may have corrupted its state while it was malfunctioning (or the actual fault may have been a bit-flip in memory—this is indistinguishable from the residue of a malfunction), and subsequent calculations based on these corrupted data will perpetuate and spread the contamination. This contamination can be arrested and repaired under a variation of the multiple-fault masking approach in which components periodically replace portions of their state with majority-voted versions from other components.

Notice that this method for transient recovery depends on a recovering component being allowed to remain an active member of the system so that it can refresh its state and obtain current sensor data. In contrast, the standard reconfiguration approach would identify a transiently-faulty component as faulty and permanently exclude it from the system. This wastes a potentially useful resource and may require additional redundancy to compensate for its loss, thereby vitiating the main benefit of reconfiguration.

In summary, multiple-fault masking can provide an attractive treatment for transient faults, but requires a lot of redundancy to mask multiple permanent faults, whereas reconfiguration provides good treatment for permanent faults, but is less attractive for transient faults. In this paper, we present a design that combines the attractive features of both these approaches with few of their disadvantages.

The paper is organized as follows. The traditional architecture for state machine replication with transient recovery is described in the next section. The proposed architecture with limited reconfiguration is described in Section 3 and its main algorithms, those for "extended" interactive consistency, are given in Section 4. The other algorithms required are outlined in Section 5 and conclusions presented in Section 6.

2 State Machine Replication with Transient Recovery

Our focus is fault tolerance through active replication of components that may exhibit Byzantine (i.e., uncontrolled or arbitrary) failures, using what is sometimes called the *state machine* approach [22], in the form introduced by SIFT [28] and subsequently refined by MAFT [11].[3]

A frame-synchronous architecture based on state machine replication operates as follows. There are n replicated major components, generally called *channels*, that are electrically and physically isolated from each other so that their failures will, to the extent possible, be independent. Each channel has a processor, some memory, and access to sensors and actuators. The channels execute the same software and are synchronized so that they perform actions at approximately the same time. At the beginning of each "frame" of activity, all channels sample their sensors and then exchange their sensor readings in a manner that ensures *interactive consistency* (also called "source congruence" or "Byzantine agreement"), meaning that all nonfaulty channels are guaranteed to have identical sets of sensor readings. All channels then execute the same sensor conditioning and selection software to generate inputs for the control laws. All channels evaluate the same control laws and their outputs are then subjected to exact-match majority voting before being sent to the actuators. In order that voting should not become a single point of failure, each channel usually has its own voter, and the voted values from each channel are then further voted or averaged at the actuators through some form of "force-summing." For example, the outputs of different channels may energize separate coils of a single solenoid, or multiple hydraulic pistons may be linked to a single shaft.

There are three key elements to this architecture; collectively they ensure and exploit *replica determinism* [19].

1. Clock synchronization ensures that all nonfaulty channels can coordinate their actions and communications using a common notion of time. Synchronization between nonfaulty channels must be achieved in the presence of disruption caused by faulty channels. The number of faulty channels that can be tolerated depends on the algorithm and fault model employed, but under the weakest assumptions it is fewer than one-third [9, 14].

2. Interactive consistency ensures that all nonfaulty channels start their computations with identical sets of sensor readings. This also must be achieved in the presence of faults; the requirements are that all nonfaulty channels have a correct view of each others' sensor readings, and a common view of the sensor readings communicated by faulty channels. As with clock synchronization, this can be achieved under the weakest assumptions only if fewer than one third of the channels are faulty. An important parameter to all algorithms for interactive consistency is the number of *rounds* of information exchange that they perform; no r-round algorithm can withstand more than r simultane-

[3] Schneider's tutorial [22] describes the state machine approach in its client-server form. Here, we use the original SIFT form (which is suited to control applications) where both clients and servers are combined into a single replicated application. Other systems that use active replication include Delta-4 [2], FTP [13], and Mars [12].

ous Byzantine faults [6, 7]. Resource constraints on communication usually restrict the number of rounds to some small fixed number—generally one.

3. Exact-match majority voting of actuator outputs masks faults in execution of the main computational tasks. Because interactive consistency ensures that all nonfaulty channels start off with the same set of sensor readings, and because they all perform identical sensor conditioning and selection and the same control law calculations, all nonfaulty channels will produce identical outputs. Therefore exact-match majority voting will mask the outputs of failed channels, provided that fewer than half the channels are faulty [28].[4]

2.1 Multiple-Fault Tolerance

Notice the different number of faulty channels tolerated by the different elements of this architecture: fewer than a third for clock synchronization and interactive consistency, and fewer than a half for the computational tasks. Thus, for example, a five-channel system can tolerate two faulty channels in the computational tasks, but only one in clock synchronization or interactive consistency. If we require a two-fault-tolerant system, then we must provide seven channels for clock synchronization and interactive consistency, and must use a two-round algorithm for the latter.

These calculations are based on worst-case assumptions about the behavior of faulty channels (i.e., the *Byzantine* fault model). It is possible to tolerate more faulty channels in clock synchronization and interactive consistency if some of their failure modes are "not too bad." *Hybrid* fault models and their associated algorithms consider both Byzantine and less serious kinds of failures occurring in combination [17, 21, 24] while *authenticated* algorithms use digital signatures to constrain the behavior of faulty channels [6, 15]. These approaches (which can be used in combination [8]) increase the total number of faults that can be tolerated in clock synchronization and interactive consistency, but Byzantine failures can never exceed the number of rounds in interactive consistency [7], nor one third of the channels in clock synchronization [9].

2.2 Transient Recovery

As noted, transient faults are many times more common than permanent faults. When a channel whose processor has been afflicted with a transient fault recovers its ability to execute programs correctly, its overall fault status depends on the extent to which its state data has been corrupted and on the task being performed.[5] A task that uses only current sensor data (which are refreshed every frame) or uncorrupted state data will be performed correctly; one that uses corrupted state data will be performed incorrectly.

Clock synchronization and interactive consistency rely on little or no state data[6] and are therefore very likely to be performed correctly by a channel whose processor has recovered from a transient fault. It follows that a number of sequential transient faults may be tolerated perfectly well by single-fault-tolerant mechanisms for clock synchronization and interactive consistency.

The main computational tasks, however, will access state data and those of them that reference corrupted data may be performed incorrectly. Results of incorrect calculations may then be stored in previously uncorrupted locations. Thus a transient fault can lead to *permanent* failure of its channel, and a sequence of such faults in different channels can exhaust the multiple-fault-tolerance provided by majority voting of the main computational tasks.

This undesirable scenario can be overcome by causing all channels to exchange their private state periodically; each channel then performs a majority vote on the exchanged data and replaces its private copy by the majority value. Since the amount of state data may be large, only portions of it are generally exchanged and voted at each stage, so that it may take several frames to refresh the entire state. In this way, a channel that is recovering from a transient fault (i.e., one whose processor is calculating correctly, but that has corrupted state) will be able gradually to repair its state. Provided that enough channels remain nonfaulty at all times, and provided state data are not contaminated faster than they are repaired, this approach provides self-stabilizing recovery from transient faults in the main computational tasks [20].

Observe that, whereas the effectiveness of fault masking by majority voting of system outputs is independent of the application tasks, the effectiveness of transient recovery depends on application properties such as the dataflow dependencies among tasks and

[4] A channel *fails* when it manifests incorrect behavior at some external interface. We say that a channel is *faulty* if there is something wrong with it; the fault may or may not lead to failure, but correct operation cannot be assumed.

[5] Programs are assumed to be stored in ROM and to be immune to corruption.

[6] Clock synchronization generally requires the local clock adjustment plus the frame counter. Oral Message algorithms for interactive consistency require nothing; authenticated algorithms can require a significant quantity of data to maintain cryptosynch and to foil spoofing attacks [8], which militates against their use in combination with transient recovery.

the pattern through which portions of state data are replaced by majority voted copies [20]. By adjusting the voting pattern to the dataflow structure of the application, recovery can be guaranteed in a fixed number of frames [5].

2.3 Residual Weaknesses

State machine replication, augmented to provide transient recovery in the manner just described, is an attractively simple architecture that provides robust fault tolerance against transient Byzantine faults that arrive (but do not necessarily clear) sequentially. However, practical concerns limit the level of redundancy and the number of rounds that can be provided, so the architecture is vulnerable to the arrival of multiple faults in close proximity. For example, a five-channel, one-round system cannot tolerate more than one active Byzantine fault in its mechanisms for interactive consistency or clock synchronization. A permanent fault, or a transient fault that is slow to clear, exposes the system to total failure should another fault arrive. This vulnerability could be reduced if the system reconfigured to exclude the faulty channel. (The resulting four-channel, one-round system can tolerate an additional Byzantine fault.) The next section introduces mechanisms to accomplish this.

3 Adding Reconfiguration: The Proposed Architecture

Our goal in extending the architecture described in the previous section is to retain most of its good features, while overcoming its weaknesses. One of its good features is the self-stabilizing approach to transient recovery: transiently faulty channels remain in the system and are thereby able to repair their state and to recover their health. However, this is also the source of the principal weakness in the architecture: because faulty channels remain active in the system, those with permanent faults, or transient faults that are slow to clear, can seriously degrade its ability to mask additional faults in clock synchronization and interactive consistency.

We could attempt to overcome this weakness by diagnosing permanently faulty channels and reconfiguring to exclude them. However, this approach would place great reliance on the accuracy and completeness of diagnosis, and on its ability to discriminate between transient and permanent faults, which is hard to justify in the presence of arbitrary (i.e., Byzantine) faults.

Our chosen approach takes a middle course between leaving faulty channels in the system and reconfiguring to exclude them completely. Channels suspected of being faulty can be placed on *probation*, which is a limbo state between full inclusion in the system and complete exclusion. Those channels that are not on probation constitute the current *core* of the system. A requirement of the architecture is to ensure that the core comprises sufficiently many nonfaulty channels that it can achieve interactive consistency and clock synchronization among its members; a desirable goal is to dimension the system so that the core can tolerate *additional* faults among its members. Core members suspected of being faulty can be moved to the probationary status. Faults among the probationers have no impact on the core, but those probationers that are nonfaulty are able to "shadow" the core so that they can establish and maintain clock and state synchrony with the core. Furthermore, probationers that happen to be nonfaulty (either because they were inaccurately diagnosed, or because they have recovered from a transient fault) are able to contribute sensor data to the core in a reliable manner. Probationers that appear to have been working correctly for some time can be readmitted to the core.

In more detail, the architecture that we propose is state machine replication with transient recovery, extended as follows.

- All channels maintain in their state a record of the current core members.
- The core members maintain clock synchronization among themselves using a standard algorithm. Nonfaulty probationers maintain synchronization by "listening in" to the execution of this algorithm (in a manner described in Section 5.1).
- The exchange of single-source data such as sensor samples and diagnostic information is performed using an algorithm for "extended" interactive consistency, as described in the next section.
- All channels contribute to the majority voting of actuator outputs and participate in the periodic exchange of state data and update their own state data with majority voted values.
- All channels periodically exchange diagnostic "syndrome" information in an interactively consistent manner and use this to modify their estimate of those channels that should constitute the core.

This architecture successfully masks faults provided no more than a minority of channels are faulty at any one time, and provided the core is always able to mask faults among its own members in clock synchronization

and interactive consistency. The latter can be achieved using standard algorithms for clock synchronization and (one-round) interactive consistency as long as the core always contains at least four members, and provided faulty members are diagnosed and excluded before the next fault arrives. This allows $n-3$ arbitrary faults to be tolerated at any time, provided they arrive sequentially, so that clock synchronization and interactive consistency are no longer limiting factors on the number of faults that can be tolerated.[7]

This extended architecture depends on algorithms for *diagnosis* (to identify faulty and recovered channels) and *group membership* (to coordinate membership of the core), and on modified treatments of clock synchronization and interactive consistency (to allow probationers to participate without disrupting the core). It is the modified treatment of interactive consistency that is central to the development of the proposed architecture, so we introduce it first.

4 Extended Interactive Consistency

We begin by describing the classical problem of Interactive Consistency, and the Oral Messages algorithm. We then describe the extended problem and modified algorithms.

4.1 Classical Interactive Consistency

In the classical problem of *Interactive Consistency* [18], there are n processors, of which some number t may be faulty. Each processor p has some private value (e.g., from sampling a sensor) ν_p and the goal is to distribute these private values to all the other processors so that each processor q can form its estimate $V_q(p)$ of processor p's private value, satisfying the following conditions:

Agreement: If processors q and r are nonfaulty, then they agree on the value ascribed to any other processor p; that is: $V_q(p) = V_r(p)$.

Validity: If processors p and q are nonfaulty, then the value ascribed to p by q is indeed p's private value; that is, $V_q(p) = \nu_p$.

Interactive consistency is the symmetric version of *Byzantine Agreement*: in the latter problem, there is a distinguished processor called the *transmitter* (it is called the "Commanding General" in the metaphor of the Byzantine Generals [15]) and the goal is to distribute its private value to all the other processors (which are called *receivers*) in a manner that ensures Agreement and Validity. An algorithm for Byzantine agreement can be converted to one for interactive consistency by simply iterating it over all processors (each in turn taking the role of the transmitter). It is usually simpler to describe algorithms in their Byzantine agreement form, and we will follow this practice here.

Algorithms for interactive consistency can be developed under various *fault models* and *message assumptions*. The most robust of these are the *Byzantine* fault model (there are no assumptions on the behavior of a faulty processor), and *Oral Messages* (faulty processors can manufacture messages purporting to come from other processors, and can change any messages that they forward from one processor to another). The r-round algorithm OM(r) can solve the problem under these assumptions, provided $r \geq t$ and $n > 3r$ [15].

The algorithm is described recursively; the base case is OM(0).

1. The transmitter sends its value to every receiver.

2. Each receiver uses the value obtained from the transmitter, or some arbitrary, but fixed, value if nothing is received.

Next, we describe the general case, OM(r), $r > 0$.

1. The transmitter sends its value to every receiver.

2. For each p, let v_p be the value receiver p obtains from the transmitter, or else be some arbitrary, but fixed, value if it obtains no value. Each receiver p acts as the transmitter in Algorithm OM($r-1$) to communicate this value v_p to each of the $n-2$ other receivers.

3. For each p, and each $q \neq p$, let v_q be the value receiver p obtained from receiver q in step (2) (using Algorithm OM($r-1$)), or else some arbitrary, but fixed, value if nothing was received. Each receiver p calculates the majority value among all values v_q it receives, and uses that as the transmitter's value; if no absolute majority exists, then p uses some value functionally determined by the set of v_q's.

This algorithm is optimal within its class: all algorithms based on oral message assumptions require at least $3t+1$ processors and t rounds to tolerate t Byzantine faults. Other algorithms, based on different assumptions, can withstand more faults for the same number of processors but none can do so in fewer rounds.

[7] The need to maintain an overall majority of nonfaulty channels limits the number of faults to $\lfloor \frac{n-1}{2} \rfloor$.

4.2 The Extended Interactive Consistency Problem

In the Extended version of Interactive Consistency, there are again some number n of processors, but this time they are partitioned into p *probationers* and c *core members* (so that $n = p + c$); some number t of core members are faulty, and possibly all of the probationers. The goal is to exchange private values among all processors (both probationers and core members) in a manner that satisfies both Agreement and Validity. Thus the Extended problem is like classical Interactive Consistency, except that we have to tolerate as many as $t + p$ faults, given that no more that t of them are among the core members. The basic idea is that if core members are able to tolerate the t faults among themselves, then they can act as a fault-tolerant distribution mechanism to and from the probationers.

In the following subsections, we describe modifications to the classical OM algorithm that solve the extended interactive consistency problem; later, we briefly consider similar modifications to other interactive consistency algorithms.

4.2.1 Extended Oral Messages. The extended OM algorithms are described in their Byzantine agreement formulation (i.e., where there is a distinguished transmitter). We begin with the case where the transmitter is one of the core members.

The Majority Vote Algorithm. In this algorithm, the core members run OM(r) among themselves. They will achieve Validity and Agreement provided $r \geq t$ and $c > 3r$. Each member of the core then sends its estimate of the transmitter's value to each probationer (the transmitter sends its own value). The inequalities above ensure that a strict majority of the core members are nonfaulty (i.e., $c > 2t$). Each nonfaulty probationer can therefore perform a majority vote on the values that it receives in order to decide on the same value as that used by the nonfaulty members of the core. □

A disadvantage of this algorithm is that it requires an additional round of broadcast messages (from each member of the core to each probationer) following those required for OM(r). Because message rounds are the critical resource in many implementations of state machine replication, this reduces the utility of the algorithm. Fortunately, it is possible to avoid this extra cost in the important case $r = 1$.

Optimization for $r = 1$. Here, the transmitter sends its value to core members only. Each member of the core then sends the value received to all other core members and to the probationers. Each receiver (both core members and probationers) then selects the majority value among those received. The key to this algorithm is that the probationers do not receive a value directly from the transmitter.

For core members, this algorithm is just OM(1), and it works as long as $1 \geq t$ and $c > 3$. For probationers, the argument goes as follows. If the transmitter is nonfaulty, each probationer gets at least $c - 2$ good values and at most one bad one (since there can be at most one faulty processor among the core). The good value wins the majority vote provided $c - 2 > 1$ (i.e., $c > 3$). If the transmitter is faulty, then all the other core members must be nonfaulty, and they will send the probationers the same values as they send each other. Thus both core members and probationers receive identical sets of values in the forwarding round. Although no majority need exist among these values, the selection algorithm is deterministic, so all receivers will decide on the same value. □

Next, we consider the case where a probationer is the transmitter. This capability is needed in deciding whether to readmit a probationer to the core (the existing core members must agree on the diagnostic syndrome data provided by the probationer), and also provides a means to obtain samples from sensors attached to a specific probationer. As with the previous case, we propose a straightforward extension to the existing algorithm.

The Extra-Round Algorithm. Assume the basic algorithm is OM(r). The transmitter (a probationer) sends its value to every core member. The core members then run OM(r) on those values to achieve interactive consistency on the set of values received, and each selects the majority value in that set (or some value functionally determined by the set if no majority exists). These values are then distributed to the other probationers as in the majority vote algorithm.

We prove that this algorithm achieves Agreement and Validity provided $r \geq t$ and $c > 3r$. Observe that the algorithm is actually the same as OM($r + 1$) on the $c+1$ processors comprising the core plus the transmitter, followed by distribution to the probationers as in majority vote algorithm. There may be as many as t faults among the core, and the transmitter may be faulty, so to achieve interactive consistency among the core we need to withstand $t+1$ faults with $r+1$ rounds among $c + 1$ processors. We have $r \geq t$ and $c > 3r$, which ensures $r + 1 \geq t + 1$, but not $c + 1 > 3(t + 1)$ as required for OM($r + 1$). That is, we have enough

rounds, but possibly insufficient processors to mask the number of faults that may be present. With insufficient processors, OM($r+1$) can fail in circumstances where OM(r) succeeds (consider $r = 1$ and four processors with a Byzantine fault among the receivers), so further analysis is required.

We have $c > 3t$, so there are sufficient processors among the core to reach interactive consistency on the value received from the transmitter by each member of the core. If the transmitter is nonfaulty, the nonfaulty core members will each receive the correct value and this will win the majority vote provided $c - t > 2t$ (there are at least $c - t$ nonfaulty core members and at most t faulty ones) and this follows from $c > 3t$. If the transmitter is faulty, we only require Agreement and this follows because the nonfaulty members of the core each have the same set of values and the selection process is functionally determined by this set.

Correctness of the step that distributes the value from the core to the other probationers follows by the argument used for the majority vote algorithm. □ .

This algorithm has the disadvantage of requiring two extra rounds of messages: one to distribute the transmitter's value to the core, and one to distribute the chosen value from each core member to each probationer. There seems no way to eliminate the first of these because, in the worst case, we need to tolerate $t+1$ faults (the transmitter plus t of the core members) and this cannot be done in fewer than $t+1$ rounds. The extra rounds for the distribution to the probationers algorithm can be eliminated when the basic algorithm is OM(1) using the $r = 1$ optimization described earlier.

Summary. Interactive consistency algorithms that use majority voting, such as OM, can be extended to tolerate an arbitrary number of faults, provided there are no more than t faults among those components identified as the core, where t is the number of faults tolerated by the basic algorithm. Notice that it essential that nonfaulty components (probationers as well as those in the core) have accurate knowledge of the membership of the core, but does no harm if other (faulty) channels "think" they are in the core.

In the case of one-round algorithms (where one fault can be tolerated in the core, provided this contains at least four members), there is no additional cost to the extended algorithm when the transmitter is a member of the core. If we can do without samples from sensors attached to probationers, then the only time that a probationer needs to be the transmitter in interactive consistency (thereby incurring the cost of an additional round of information exchange) is when it provides its diagnostic syndrome information (with a view to being readmitted to the core). This can probably be done less often than every frame, thereby minimizing its cost.

4.2.2 Other Interactive Consistency Algorithms. Interactive consistency algorithms derived from OM, such as OMH for the hybrid fault model [17], and ZA for the authenticated case [8], can be extended in exactly the same way as OM, since they are also based on majority voting (albeit in modified form). It is not clear whether algorithms such as the classical Signed Messages algorithm SM [15] that do not use majority voting can be extended in a similar manner.

5 Clock Synchronization, Diagnosis, and Group Membership

In this section, we briefly describe issues in these additional elements of the architecture.

5.1 Clock Synchronization

Unlike interactive consistency, where it can be necessary for probationers to participate fully (i.e., as transmitters), clock synchronization can use an unequal arrangement. The idea is that the core members synchronize among themselves using an algorithm such as interactive convergence [14], ignoring inputs from probationers (in the case of interactive convergence, this essentially means treating their clock differences as if they are zero). Probationers run the algorithm *as if* they were members of the core (in the case of interactive convergence, this means they ignore clock differences from other probationers and base their own corrections on a fault-tolerant average of their clock differences with the members of the core).

Provided the core contains enough channels to mask faults among its own members, this arrangement allows all nonfaulty channels to remain synchronized with the core. That is, the skew between the clock of any nonfaulty channel and that of any nonfaulty member of the core will be less than some parameter δ. The skew between two nonfaulty probationers may therefore be as great at 2δ, and the system timing must allow for this looser synchronization.

5.2 Diagnosis

Diagnosis of faulty components in state machine architectures can be based on the observation that all nonfaulty channels perform identical calculations on identical data, and must therefore generate identical outputs. Majority voting is sufficient to mask faults

among a minority of channels, and any channel whose output differs from the majority must be faulty. If the channels exchange their output values, any nonfaulty channel that receives a value different than its own from some other channel may correctly conclude that the other channel is faulty. This approach is complicated, however, by the need to achieve consensus (in order to decide membership of the core) and by the fact that Byzantine faulty channels may send correct values to some channels and incorrect ones to others.

Diagnosis becomes consistent (all nonfaulty channels agree on the fault status of all channels) and accurate (no nonfaulty channel is declared faulty) if an interactive consistency algorithm is used to distribute the output values (subject to number of faults being within the tolerance of the algorithm, number of channels, and number of rounds concerned). However, this method of diagnosis is not complete: not all faulty channels will be identified. For example, if a faulty channel sends an incorrect value to one nonfaulty channel and the correct value to all others, then the receiving channel will report the incorrect value but the other channels will have insufficient evidence to decide whether this is the correct report of an incorrect value or the incorrect report of a correct one and will be unable to conclude which of the two channels is faulty.

It is possible to improve the completeness of diagnosis by accumulating evidence over time. For example, if channel a reports that channel x sent an incorrect value in one frame, and channel b that it sent an incorrect value in another, then we have two accusations against x, which is sufficient to accurately diagnose it as faulty under a single-fault assumption. Several authors develop several sophisticated diagnosis algorithms of this type [1,26,27]; the accuracy of some of these has been formally verified by Lincoln [16].

This approach can be inaccurate in the presence of transient faults, however. For example, in the scenario just given, a and b may be transiently faulty in the (different) frames when they accuse x of sending an incorrect value. Although no more than a single fault is active at any one time, the nonfaulty channel will be inaccurately diagnosed as faulty. In an architecture where reconfiguration permanently excludes channels diagnosed as faulty, inaccurate diagnosis can be very harmful (since it leaves a faulty channel operating while removing a nonfaulty one).

In the architecture proposed here, the consequences of inaccurate diagnosis are less severe, because a channel diagnosed as faulty is merely placed on probation and can be readmitted to the core if it subsequently appears nonfaulty. In these circumstances, it may be preferable to trade accuracy for completeness and to quickly place a merely suspect channel on probation, where it can be kept under surveillance or made to undergo self-tests or a reboot, rather than to favor accuracy and to leave it in the core with the hope of eventually obtaining unequivocal evidence of its faultiness. Lincoln describes a diagnosis algorithm called UD that favors completeness over accuracy [16].

We have argued previously that diagnosis should not be considered separately from reconfiguration [4]: it is not abstract accuracy or completeness of diagnosis that is important, but its contribution to successful reconfiguration. In the architecture proposed here, diagnosis will be used in group membership, and the crucial factor in deciding whether a channel should remain in the core is its ability to contribute to clock synchronization and to interactive consistency—which are dependent on correct operation of its processor (and clock), but not on correctness of its state data. For example, in a five channel system, we may be better off leaving a channel with a correct processor but faulty state data in the core rather than placing it on probation. (Although one-round interactive consistency algorithms can withstand a single Byzantine fault among either four or five processors, hybrid or authenticated algorithms can withstand larger numbers of nonByzantine faults among five processors than among four.) If the channel has a faulty processor, however, we are better off placing it on probation, because we will otherwise be vulnerable to failure if a second fault arrives. (Some one-round interactive consistency algorithm can withstand two simultaneous faults among five processors, but only if their fault modes are relatively benign [8,17]; no one-round algorithm can withstand two Byzantine faults.)

Diagnosis based on comparison of output values does not distinguish errors due to a faulty processor from those due to bad state data. A plausible way to distinguish these is to require each processor to perform a complex computation on some fresh item of data[8] that is common to all channels. A suitable item might be the value of the frame counter: this value should be among those that the channels exchange and subject to majority voting in each frame, and the diagnostic computation should be performed on the majority-voted value.

The goal of the diagnosis step should be to collect *syndrome* data that can be used in deciding group membership. Suitable components of the syndrome include the following.

- The results of a self-test (to identify processor faults).

[8] It should be fresh in order to detect a processor that merely reuses the result from an earlier frame.

- The results of a computation performed on recent majority-voted data, such as the frame counter (to help identify processor and communication faults).
- The output values sent to the actuators (to identify processor and state data faults).

Each channel (whether in the core or on probation) collects these components of its syndrome, and distributes them to all other channels in a manner that ensures interactive consistency. Algorithms for achieving interactive consistency in this architecture were described in Section 4. If the fault tolerance of the basic interactive consistency algorithm employed is t, and if the core contains t or fewer faulty channels, then all nonfaulty channels will obtain each others' syndromes correctly, and will agree on the syndromes ascribed to faulty channels.

5.3 Group Membership

The purpose of collecting syndrome information is to decide which channels should be in the core and which should be placed on probation. This is an instance of the *group membership* problem for synchronous systems first introduced by Cristian [3]. The present instance is simpler than the general problem because all nonfaulty channels are performing the same computations in lockstep. Thus, all that is necessary is for each channel to run an identical, deterministic, selection algorithm on the interactively consistent syndrome information that was distributed in the diagnosis phase. The selection algorithm identifies those channels that are to be members of the next incarnation of the core.

Notice that it is not necessary to restrict this computation to those channels that are in the core, nor to require interactive consistency on the results of the computation. Provided the core maintains enough redundancy to mask faults among its own members, all nonfaulty channels (including those among the probationers and no matter how many faulty probationers there are) will achieve interactive consistency on the syndrome data for all channels. By running a deterministic computation, all nonfaulty channels will then arrive at a common view of the membership of the core.

We do not consider here the actions that should be taken when a channel is first placed on probation. It may be desirable to force it to reboot its processor and to run diagnostics: this can be achieved by having all nonfaulty channels send a "reboot" signal to the channel concerned. A majority voter on the "reboot" line will ensure this is performed only when appropriate (since the nonfaulty processors must always be a majority).

6 Conclusions

The interactive consistency and clock synchronization elements of architectures based on state machine replication can generally tolerate fewer faults than the majority voting used to mask faults at the actuators. Reconfiguring to permanently exclude faulty components allows these elements to tolerate more faults, but places great demands on accuracy and completeness of the diagnosis of faulty components, and provides poor treatment for transient faults. We have proposed an architecture that allows components diagnosed as faulty to continue operation "on probation," so that transiently faulty components can be allowed to recover and to repair their state, but in such a way that they do not degrade the fault tolerance of clock synchronization and interactive consistency. The MAFT architecture [11, 25] took a similar overall approach to that proposed here, with its "operating set" being similar to our "core," but without our extended treatment for interactive consistency.

Because diagnosis is uncertain, the fault tolerance achieved by the proposed architecture is a stochastic property—unlike that of standard state machine replication, where worst-case fault tolerance has a simple characterization. We do not advocate using this architecture to reduce replication levels below those required for the standard approach, but to reduce the likelihood of failure should the number of faults exceed those for which the system was dimensioned. If diagnosis is done in a cautious way (so that only indisputably faulty components are placed on probation), the proposed approach is sure to increase fault tolerance, while adding very little complexity to standard state machine replication. The tradeoffs in using less cautious diagnosis can be evaluated experimentally or with reliability modeling techniques. In future, we hope to formally verify properties of the proposed architecture.

Acknowledgments. Thoughtful comments by the anonymous referees and by Roger Kieckhafer were very helpful in preparing the final version of this paper.

References

[1] Richard W. Buskens and Ronald P. Bianchini, Jr. Distributed on-line diagnosis in the presence of arbitrary faults. In *Fault Tolerant Computing Symposium 23* [10], pages 470–479.

[2] Marc Chérèque, David Powell, Philippe Reynier, Jean-Luc Richier, and Jacques Voiron. Active replication in Delta-4. In *Fault Tolerant Computing Symposium 22*, pages 28–37, Boston, MA, July 1992. IEEE Computer Society.

[3] Flaviu Cristian. Agreeing who is present and who is absent in a synchronous distributed system. In *Fault Tolerant Computing Symposium 18*, pages 206–211, Tokyo, Japan, June 1988. IEEE Computer Society.

[4] Judith Crow and John Rushby. Model-based reconfiguration: Toward an integration with diagnosis. In *Proceedings, AAAI-91 (Volume 2)*, pages 836–841, Anaheim, CA, July 1991.

[5] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. High level design proof of a reliable computing platform. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications—2*, volume 6 of *Dependable Computing and Fault-Tolerant Systems*, pages 279–306. Springer-Verlag, Vienna, Austria, February 1991.

[6] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, November 1983.

[7] M. Fischer and N. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14:183–186, 1982.

[8] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In *Dependable Computing for Critical Applications—5*, pages 79–90, Champaign, IL, September 1995. IFIP WG 10.4, preliminary proceedings.

[9] J. Y. Halpern, B. B. Simons, H. R. Strong, and D. Dolev. Fault-tolerant clock synchronization. In *Third ACM Symposium on Principles of Distributed Computing*, pages 89–102, Vancouver, B.C., Canada, August 1984. Association for Computing Machinery.

[10] *Fault Tolerant Computing Symposium 23*, Toulouse, France, June 1993. IEEE Computer Society.

[11] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

[12] Hermann Kopetz *et al.* Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9(1):25–40, February 1989.

[13] Jaynarayan H. Lala and Richard E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82(1):25–40, January 1994.

[14] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[15] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[16] Patrick Lincoln. Formally verified algorithms for diagnosis of manifest, symmetric, link, and Byzantine faults. Technical Report SRI-CSL-95-14, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1995.

[17] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23* [10], pages 402–411.

[18] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[19] Stefan Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6:289–316, 1994.

[20] John Rushby. A fault-masking and transient-recovery model for digital flight-control systems. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Kluwer International Series in Engineering and Computer Science, chapter 5, pages 109–136. 1993.

[21] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Los Angeles, CA, August 1994. Association for Computing Machinery.

[22] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[23] Kang C. Shin and P. Ramanathan. Diagnosis of processors with Byzantine faults in a distributed computing system. In *Fault Tolerant Computing Symposium 17*, pages 55–60, Pittsburgh, PA, July 1987. IEEE Computer Society.

[24] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, Columbus, OH, Oct. 1988. IEEE Computer Society.

[25] P. Thambidurai, A. M. Finn, R. M. Kieckhafer, and C. J. Walter. Clock synchronization in MAFT. In *Fault Tolerant Computing Symposium 19*, pages 142–149, Chicago, IL, June 1989. IEEE Computer Society.

[26] C. J. Walter, N. Suri, and M. M. Hugue. Continual online diagnosis of hybrid faults. In F. Cristian, G. Le Lann, and T. Lunt, editors, *Dependable Computing for Critical Applications—4*, volume 9 of *Dependable Computing and Fault-Tolerant Systems*, pages 233–249. Springer-Verlag, Vienna, Austria, January 1994.

[27] Chris J. Walter. Identifying the cause of detected errors. In *Fault Tolerant Computing Symposium 20*, pages 48–55, Newcastle upon Tyne, UK, June 1990. IEEE Computer Society.

[28] John H. Wensley *et al.* SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

From Dependable Computing for Critical Applications–5, Champaign, IL, September 1995, pp. 139–157; Volume 10 of the series in Dependable Computing and Fault Tolerant Systems published by IEEE Computer Society Press.

Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid and Link Faults*

Li Gong†, Patrick Lincoln, and John Rushby
Computer Science Laboratory
SRI International
Menlo Park, California 94025, USA

Abstract

We show that the assumptions required of the authentication mechanism in Byzantine agreement protocols that use "signed messages" are stronger than generally realized, and require more than simple digital signatures. The protocols may fail if these assumptions are violated. We then present new protocols for Byzantine agreement that add authentication to "oral message" protocols so that additional resilience is obtained with authentication, but with no assumptions required about the security of authentication when the number and kind of faults present are within the resilience of the unauthenticated protocol.

Our analysis is performed under a "hybrid" fault model that admits manifest (e.g., crash) and symmetric faults as well as arbitrary (i.e., Byzantine) faults. We also extend the classical signed messages protocol to this fault model, and show that its fault tolerance is matched by one of our new protocols. We then explore the behavior of these various protocols under the combination of hybrid processor faults and communications link faults. Using formal state-exploration techniques, we examine cases beyond those guaranteed by simple worst-case bounds and find that the resilience of one of the new protocols exceeds that of the others in these regions.

The new protocols are superior to other known protocols in properties and measures of practical interest, and we recommend them for general use. They are particularly attractive in security-critical systems where authentication may be subjected to sophisticated cryptographic attack, and in safety-critical embedded systems where it may be necessary to use very short signatures, but where maximum resilience is required.

1 Introduction

A fundamental requirement in fault-tolerant systems based on the "state machine" approach [27] is for replicated processors to reach agreement on the values of single-source data, such as sensor samples. In its abstract form, this is the problem of Byzantine Agreement (and its variant, the problem of "Interactive Consistency," also known as "source congruence," "distributed consensus," and "reliable multicast") [16, 23]. There are two broad classes of protocols for achieving Byzantine agreement. Those based on "oral message" assumptions place no restrictions on what a faulty processor may do; those based on "written message" assumptions disallow faulty processes making undetectable modifications to messages as they are relayed from one processor to another, and also disallow processors manufacturing messages that purport to come from another processor. It is generally stated that the written messages assumptions can be satisfied using cryptographic authentication methods (i.e., "digital signatures"), and protocols based on these assumptions are therefore often called "signed messages" or "authenticated" protocols [5, 11, 16].

Both oral and written message protocols proceed in "rounds" and the parameters of interest include: how many faults can be tolerated by a given number of processors, and how many rounds and how many messages are required? Theoretical studies also consider the size of the messages, or the total number of bits transmitted. The advantage of written messages protocols is that they can generally withstand more faults than oral message protocols, and often require fewer messages. For example, oral message protocols require $3t + 1$ processors to withstand t faults, while written messages protocols require only $t + 2$ (the problem is vacuous unless there are at least two nonfaulty pro-

*This work was supported in part by the National Aeronautics and Space Administration, Langley Research Center, under contract NAS1-20334, by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044, and by the National Science Foundation under contract CCR-9509931.

†Li Gong is now with JavaSoft and can be reached at gong@eng.sun.com

cessors). However, both classes of protocols provably require $t + 1$ rounds in the worst case [5, 11], though "early stopping" protocols (which are most easily constructed under the written messages assumptions) use fewer rounds when the actual number of faults is less than t [2, 7, 8, 10, 12].

It would seem that the written messages protocols have significant advantages over their oral message counterparts (e.g., asymptotically, a three-fold advantage in number of faults tolerated). However, these advantages may not be so significant in practice. In embedded applications, the most severe practical constraint on these protocols is the number of rounds: a given application will generally fix the number r of rounds it can afford (generally two). This, in turn, fixes the number of faults that can be tolerated at $r-1$, independently of the class of protocols chosen.[1] The class of protocols does affect the number of processors required: e.g., two-round written message protocols require three processors to tolerate a single fault, while oral message protocols require four. But if other purposes (e.g., clock synchronization) already require four or more processors, there seems no compelling reason to use written message protocols. In fact, there is an argument against these protocols which Chris Walter, one of the developers of the MAFT architecture for fault-tolerant flight control [15] expressed to us as follows: "you have to assume that digital signatures satisfy the requirements for written messages, and in life-critical systems we prefer to make as few assumptions as possible." It turns out that this caution is justified.

In the rest of the paper, we first describe the various assumptions that such protocols (we will call them "authenticated protocols") depend on, highlighting the risks in placing the correctness of Byzantine agreement on the effectiveness of cryptographic protocols for which currently there is no method of assurance that is definitive and generally accepted. We note, however, that authenticated protocols can tolerate more faults than oral message protocols, and we show that this advantage is retained when the analysis is extended to a hybrid fault model that counts faults more carefully than the purely Byzantine fault model.

We then consider the addition of authentication to variants of the Oral Messages protocol and show that this increases the number of faults they can tolerate if the assumptions on the authentication mechanism are warranted, without compromising their innate fault tolerance if those assumptions are violated. Assuming authentication, we show that one of these new protocols can tolerate as many hybrid faults as the classical Signed Messages protocol.

We then examine the two-round versions of the various protocols under an enlarged fault model that includes communications link faults. For many applications, this is the most realistic class of protocol and fault-model, and we provide evidence, derived from formal state-exploration techniques, that one of the authenticated oral message protocols provides the greatest fault tolerance.

2 Byzantine agreement, fault models, and message assumptions

In the classical Byzantine Generals problem, there are a number of participants, which we call "processors." A distinguished processor, which we call the *transmitter*, possesses a value to be communicated to all the other processors, which we call the *receivers*. (These correspond to the "Commanding General" and "Lieutenant Generals," respectively, in the terminology of Lamport, Shostak, and Pease [16].) It is assumed that there are point-to-point communications paths between each pair of processors. The Byzantine Agreement problem can be studied under several different sets of assumptions. We consider both "Oral" and "Written" message assumptions, and a "Hybrid" fault model. The *Oral Messages* assumptions are the following [16, p. 387].

A1: Every message that is sent between nonfaulty processors is correctly delivered.

A2: The receiver of a message knows who sent it (assumption of private channels).

A3: The absence of a message can be detected (assumption of synchrony).

Written Messages assumptions add the following to those of oral messages [16, p. 391].

A4(a): Messages sent by a nonfaulty processor (under the hybrid fault model—see later—this becomes a non-arbitrary-faulty processor) cannot be altered or manufactured by other processors.

A4(b): Any nonfaulty receiver can identify the processor that originated a message, if that processor is nonfaulty (again, under the hybrid fault model this becomes a non-arbitrary-faulty processor). Note that A2 concerns the case of a direct path from sender to receiver, whereas A4(b) concerns a message from an "originating sender"

[1] The small number of rounds and the deterministic processor and communications scheduling used in embedded applications also obviate the benefits of early stopping.

that is possibly relayed by other processors before reaching the receiver.

There are n processors in total, of which some (possibly including the transmitter) may be faulty. In the classical Byzantine Generals problem, there are no constraints other than those given above on the behavior of faulty processors. This leads to pessimistic estimates of the number of faults that can be tolerated because all faults are regarded as the worst possible. We therefore consider a "hybrid" fault model (originally due to Thambidurai and Park [29] and also investigated by Walter, Suri, and Hugue [30]) that distinguishes certain simpler kinds of fault as well as those that are unconstrained. The fault modes we distinguish for processors are *arbitrary-faulty*, *symmetric-faulty*, and *manifest-faulty*. A manifest fault is one that can be detected by mechanisms present in all nonfaulty processors (e.g., missing or improperly formatted messages). The other two fault modes yield behaviors that are not detectably bad: a symmetric fault presents the same faulty behavior to every nonfaulty processor; an arbitrary fault is completely unconstrained (i.e., Byzantine) and may present (possibly) different aberrant behaviors to some nonfaulty processors, and good behavior to others.

The above characterization of the hybrid fault model is a generic one; for Byzantine agreement, the characterization of fault modes has to be refined in terms of the processor behaviors relevant to this problem (see [26] for a different characterization in terms relevant to clock synchronization). The basic step in an agreement protocol is for a processor to transmit a value v to several other processors. The interpretation of a manifest fault in this context is one that produces detectably missing values (e.g., timing, omission, or crash faults), or that produces a value that all nonfaulty recipients can detect as bad (e.g., it fails checksum or format tests). Symmetric faults deliver *wrong*, rather than missing or manifestly corrupted values—but do so consistently, so that all receivers of a given transmission obtain the *same* wrong value $v' \neq v$. Arbitrary faults are unconstrained, and can deliver correct, wrong, or manifestly faulty values in any combination.

Under these assumptions, the Byzantine Agreement problem is to devise a protocol that will allow each receiver p to compute an estimate ν_p of the transmitter's value satisfying the following conditions:

Agreement: If receivers p and q are nonfaulty, then they agree on the value ascribed to the transmitter—that is, for all nonfaulty p and q, $\nu_p = \nu_q$.

Validity: If receiver p is nonfaulty, the value ascribed to the transmitter by p is

- The value actually sent, if the transmitter is nonfaulty or symmetric-faulty,
- The distinguished value E, if the transmitter is manifest-faulty.

All the Byzantine agreement protocols we consider proceed in rounds: in the first round, the transmitter sends a value to all the other processors; in subsequent rounds, these processors exchange the values received among themselves in order to detect inconsistencies; each receiver then decides on one value among those received and exchanged. How this decision is made, and how the exchanges are done, depends on the protocol considered.

Notice that the additional assumptions for written messages essentially constrain the behavior of symmetric- and arbitrary-faulty receivers: under oral message assumptions, such receivers can alter or manufacture messages purporting to come from other processors in the later rounds—this is prohibited under written messages assumptions. Authenticated protocols attempt to satisfy the written messages assumptions using digital signatures: each processor signs the messages that it sends. Any receiver can check the authenticity of a message and confirm the identity of its claimed originator by checking the signature. There are several digital signature schemes that provide these basic properties [4, 9, 22, 25]. However, in the following section we show that these schemes must be used very carefully.

3 Authentication issues

The messages that are passed among the processors in authenticated protocols have the form $\{\{\ldots\{v\}_p\ldots\}_q\}_r$ which symbolizes the value v in a message signed and sent by processor p, received signed and forwarded by processors $\ldots, q$ and finally received, signed and forwarded by processor r. If processor p is nonfaulty, then at no stage in the protocol should there exist $\{\{\ldots\{v'\}_p\ldots\}_q\}_r$ in which $v \neq v'$. (This follows because if p is nonfaulty, it would not send out two different values v and v', and authentication prevents any other processor manufacturing such a value.) It is generally assumed that this requirement is satisfied if digital signatures are simply computed on and attached to the messages being relayed. This would be true if a valid message of the form $\{\{\ldots\{v\}_p\ldots\}_q\}_r$ could only arise once in the lifetime of the protocol. Theoretical examinations of these protocols normally consider only a single "run,"

but in practice they will be called repeatedly (e.g., to distribute sensor samples at the beginning of every process control cycle). It follows that processor r could save a valid message $\{\ldots\{v'\}_p\ldots\}_q$ from one run of the protocol and could then inject the correctly signed message $\{\{\ldots\{v'\}_p\ldots\}_q\}_r$ into a later run, which will cause any nonfaulty receiver to conclude that the original sender p must be faulty, and thereby defeat the protocol.

We do not need to postulate active, intelligent attacks to be concerned about this kind of problem: a hardware "off by one" fault that causes a message to be picked up from the wrong buffer when two agreement protocols are in operation simultaneously (as when all processors are exchanging sensor data) could produce this behavior. A solution to this particular problem is to include additional information under the digital signatures that will identify messages as "fresh" (Lamport, Shostak, and Pease suggest sequence numbers [16, page 400]), but this needs to be done carefully in order to distinguish *this* run of the protocol from others that may be active simultaneously.

In the rest of this section, we discuss this and a number of other issues requiring care in the implementation of authenticated Byzantine agreement protocols.

Signature permutation. The signature system must not be commutative. Otherwise, $\forall p, q, v, \{\{v\}_p\}_q = \{\{v\}_q\}_p$ and, if the session initiator is faulty, another faulty processor can falsely accuse a third, but correct, processor of being faulty in a several-round protocol.

Verifying signature sequences. Verifying a sequence of t signatures is not trivial. A recipient can try all possible sequences of t out of n signatures, but this requires an exponential amount of computation. Or the message can include a hint, such as the identity of the signer, in each stage of the signing, so the message may look like $\{q, \{p, v\}_p\}_q$. We can alternatively require that a list of hints is attached to each message outside the signatures. However, such hints will add $O(n \log n)$ bits to the message length (in an n-round protocol), thus exceeding the tight lower bound on message bits by Srikanth and Toueg [28, Theorem 1] by a factor of n. (In today's practice, a secure digital signature uses about 512 to 1024 bits.) Note that hints are necessary whether the signature system used is commutative or not. A third approach is to globally order the messages so that a recipient can deduce from the context which signature sequence should be used for verification.

Processors are assumed to know each others' signature keys. Borcherding [3] investigates the case where there is no central authority to distribute these keys, and proposes the notion of "local authentication" to achieve a weaker version of Byzantine agreement.

Distinguishing concurrent sessions. When multiple sessions can execute at the same time, it is vital to determine to which run a message belongs. Otherwise, suppose each processor maintains a different sensor and all processors are trying to agree on the values of all sensors, then a faulty processor may "borrow" a signed message from one run and use it in another. Even a benign processor can possibly make such a mistake, as we described previously. One solution is to attach a session identifier, possibly the identity of the session initiator, to the sensor value. This solution will increase the size of each message by $O(\log n)$ bits. This does not exceed the lower bound by Srikanth and Toueg [28] because they already allocate $O(\log n)$ bits for signatures.

Detecting replay attacks. Beside distinguishing concurrent sessions initiated by different processors, it is equally important to detect any attempt to reuse past messages (from the same initiator) in a new run. The initiator must securely attach a freshness identifier to the signed value. For example, the initiator can sign both the freshness identifier and the value in the same signature.

There are three types of freshness identifiers, each of which can be used in more than one way [13]. The first is a timestamp, if processors have synchronized clocks. In this case, the initiator attaches the reading from the local clock to the value before signing them. A recipient rejects any message with a timestamp that is outside an agreed time window relative to the recipient's local clock. A significant risk exists when a faulty processor can also have a faulty clock so that the processor sends out values signed with timestamps in the future. Even if this processor were to recover, another faulty processor could play back such a message when the correct time comes. The significance of this attack lies in the fact that there is no guarantee that any correct processor will know the existence of previously signed messages (with future timestamps). To invalidate such messages, a repaired processor can change its signature key during reintegration.

The second type of a freshness identifier is a random number, also known as a "nonce." Since the nonce must be generated by the processor that is checking for freshness, processors must exchange nonces with each other (thus adding one round to the protocol),

and the value must be signed with all $O(n)$ nonces, thus increasing the message length significantly.

The third type is a counter value. Each processor maintains a monotonic counter, increments the counter value before initiating a session, and then signs the value together with the current counter value. Each processor also maintains a vector timestamp, noting the last seen counter value from every other processor, and rejects any value signed with a past counter value. Similar to timestamps, a faulty processor may sign "future" counter values, so it is prudent to change to a new signature key after repair.

Repair and restart. When a processor fails, it may lose all its state information, including the current session and round numbers and freshness identifiers. If the failure is arbitrary, then the surviving state information may be wrong. For example, its clock or counters may be turned back or forward. Moreover, simply asking every processor to reset their counters to zero is vulnerable to replay attacks. Therefore, to restore the synchrony between processors after repair, a repaired processor must use challenge-response (with nonces) to obtain from other processors fresh replies containing the current state information. Given the additional need of assigning a new signature key to the restarting processor and notifying all other processors of the corresponding public key, restart can be costly.

Message redundancy. A message containing the value to be signed must contain sufficient redundancy to protect against forgery. For example, a faulty processor p may choose a random number x and broadcast it as $\{v\}_p$ for some value v. Because it is quite possible that there is a value v' such that $x = \{\{v'\}_q\}_p$, p may effectively forge a signature of value v' signed by q. Or the faulty processor p can simply copy $\{v'\}_q$ from a previous protocol run and broadcast $\{\{v'\}_q\}_p$. Any processor r who further signs $\{\{v'\}_q\}_p$ is also spoofed.

There are many ways to introduce redundancy into the messages. One is to attach a checksum of a sufficient length to the original value. The size of the message will thus increase, perhaps by 128 bits (the size of a typical one-way hash function output) or at least $O(\log n)$ bits. Note that including a unique identifier of the current run does not provide sufficient redundancy because a randomly selected value x can be of the form $\{id, v\}_q$, and if id is for a future run, an attack can still happen in the future.

3.1 Practical implications

We have shown that authentication using digital signatures needs to be managed very carefully if it is to be secure against attack. How significant are these threats? There are two main classes of applications for authenticated Byzantine agreement protocols: secure systems that must maintain coordination in the face of capture and active subversion of system components (e.g., the AT&T "Rampart" architecture [24]), and safety-critical embedded control systems (e.g., the MAFT architecture for aircraft flight control [15]). Sophisticated cryptographic and other attacks are a given in the first class of applications, so our concern about the security of authentication needs no further justification here (the literature is replete with broken cryptographic protocols [1, 21]).

Intelligent malicious attack is not considered a serious possibility in embedded systems, and the argument in these cases is a little different. Byzantine-resilient architectures are attractive in these contexts because they simplify the case for assurance and certification: instead of a collection of fault-tolerance mechanisms to counter specific failure modes, and for which it is necessary to provide evidence of coverage and noninterference, we have a single mechanism that can withstand *any* kind of fault, up to some number, and it is only necessary to provide evidence for correctness and for the estimated overall fault arrival rate. Written message protocols compromise the purity of this position: faulty processors can no longer do absolutely *anything*, but are constrained by certain assumptions. Real processors *can* do absolutely anything when faulty, and in implementations using signed messages, it is the authentication mechanism that constrains them within the assumed fault mode. For certification, it is therefore necessary to provide strong evidence that the authentication mechanism does accomplish this: broken authentication is not just another fault to be tolerated, it is a violation of the assumptions under which correctness of the protocol—and hence of the entire architecture—is established.

We have seen that cryptographically strong authenticated protocols require even small data messages to be encapsulated in large signature and freshness-indicating wrappers, and to carry various key-management indicators. Hence, embedded systems may prefer to dispense with truly secure authenticated protocols and to use short keyed checksums (Lamport, Pease, and Shostak suggest a suitable checksum algorithm [16, page 400]), with fixed keys and simple sequence-numbers to indicate freshness. The authentication assumptions may sometimes fail to hold in this arrangement. In the following sections we present and study protocols that take advantage of authentication if it is present, but that retain Byzantine resilience even when signatures may be

forged. Since checksums will only rarely be "forged" by random malfunctions, these protocols are very well suited to the needs of embedded systems.

The discussion has so far focussed on authentication failure in one direction: failure to adequately constrain the behavior of a faulty processor. Authentication can also fail in the other direction: causing good messages to be rejected as bad. There are two ways this can come about: the authentication mechanism may be algorithmically incorrect or nonrobust (e.g., vulnerable to loss of crypto-synch), or a hardware fault might damage a key. The issues enumerated earlier in this section are intended to help designers avoid the first of these dangers; the second is more likely, but less serious, because it *is* just another fault, and will be tolerated to the same extent as other faults.

4 Signed messages with hybrid faults

We have argued that great care in implementation is necessary in order to satisfy the assumptions of the authenticated protocols. This care would be justified if the authenticated protocols had significant advantages over oral message protocols. However, for the case of practical importance—that is, two-round protocols—there appears little to choose between the two classes of protocols: the signed message protocol SM(1) and the oral messages protocol OM(1) of Lamport, Pease, and Shostak [16] both require two rounds[2], and both tolerate only a single arbitrary fault. The difference is that OM(1) requires four processors, while SM(1) requires but three. However, a variation on OM(1) called OMH(1) [19] that operates under the Hybrid fault model can tolerate a arbitrary, s symmetric, and m manifest faults simultaneously, provided n, the number of processors, satisfies $n > 2a+2s+m+1$ and $a \leq 1$. Thus, OMH(1) appears to tolerate *more* faults than SM(1) under certain circumstances. Of course, this comparison is unfair because the analysis for OMH(1) considers the hybrid fault model, whereas that for SM(1) treats all faults as arbitrary. So one item that warrants examination is the behavior of SM(1) under the hybrid fault model.

The classical signed messages protocol, SM(r) proceeds as follows [16, p. 391]:

SM(r)

> The transmitter sends a signed message to each receiver. Each receiver adds its signature to the message and sends it to the other receivers who add their signatures and send it to the others, and so on for r rounds. When all the exchanges are completed, each receiver discards any improperly signed messages, extracts the values sent by the transmitter from those that remain and applies a deterministic choice function to those values.

Note that if the transmitter is not arbitrarily-faulty, the set of values considered in the choice will be a singleton. Lamport, Pease and Shostak show [16, Theorem 2] that SM(r) can tolerate up to r faulty processors, the optimal result [6, 11].

To extend SM(r) and its analysis to the hybrid fault model is straightforward: the hybrid protocol SMH(r) simply recognizes and discards manifest-faulty values. Authentication prevents symmetric-faulty receivers from injecting correctly signed new values, so these receivers either duplicate other messages (which is harmless), or they introduce incorrectly signed messages, which will be discarded. Thus, messages from both manifest- and symmetric-faulty receivers either duplicate existing values or are ignored; hence they play no part in the protocol and it is as if these processors were absent. It follows that only arbitrary-faulty processors need be counted in the fault-tolerance calculation. Thus, by direct analogy with the corresponding result (Theorem 2, page 393) in [16], we have the following result.

Theorem 1 *For any r, Protocol SMH(r) satisfies Validity and Agreement provided $r \geq a$, where a is the number of arbitrary-faulty processors.*

The result is somewhat vacuous unless there are at least two nonfaulty processors, so we also have $n > a + s + m + 1$, and $r \geq a$. This may be compared with OMH(r), where we have $n > 2a+2s+m+r$ and $r \geq a$.

It can be seen that OMH(r) and SMH(r) have the same fault tolerance with regard to rounds, but that SMH(r) requires considerably fewer processors than OMH(r) (or, equivalently, can tolerate more faults for a given number of processors). However, this increased fault tolerance is obtained at the cost of depending on authentication: if the authentication assumptions fail for any reason, then SMH(r) may fail altogether.

5 Combining authentication and oral messages

The idea of examining SM(r) under the hybrid fault model suggests the dual inquiry: examining oral message protocols in the presence of authentication. It turns out that this yields protocols that combine the advantages of the two classes of protocols with few

[2]The parameter r to these protocols starts at zero, so that the number of rounds is $r + 1$.

of their disadvantages. As noted in the discussion of SMH(r), authentication turns symmetric-faulty receivers into manifest-faulty ones: they can only generate messages that are improperly signed. In order to exploit this in an oral messages protocol, we need a protocol that has the capability to discard bad messages. The classical protocol OM(r) does not do this, but our hybrid protocol OMH(r) does. It therefore seems the most promising place to start.

The protocol OMH(r) [19] is our modified and formally verified [17] version of Thambidurai and Park's protocol Z(r) [29], which is in turn a modification of the $r+1$-round oral messages protocol OM(r) of Lamport, Shostak, and Pease [16]. The key idea in both Z(r) and OMH(r) is to introduce a distinguished value E to record receipt of manifest-faulty messages. E values are ignored in the majority vote that each processor uses to decide its final value. In Z(r), E is used to record both manifest-faulty messages and the *report* of such messages relayed by another processor. This leads to confusion when there is a manifest-faulty transmitter and an arbitrary- or symmetrically-faulty receiver; Z(1) can fail in this circumstance, and this leads to more complex failures in the $r > 1$ cases. OMH(r) repairs this problem by treating the report of manifest-faulty values differently than those values themselves: $R(E)$ indicates the report of E, $R(R(E))$ the report of a report, and so on. An inverse function UnR is used to "strip off" these Rs at a later stage in the protocol. Only E (not $R(E)$, $R(R(E))$, etc.) is ignored in the majority vote.

As noted in the previous section, OMH(r) is able to tolerate a arbitrary, s symmetric, and m manifest faults simultaneously, provided n, the number of processors, satisfies $n > 2a + 2s + m + r$ and $r \geq a$. This is optimal when only arbitrary faults are present (we have $a = r$, $s = m = 0$, so that $n > 3a$, satisfying the lower bound established by Pease, Shostak, and Lamport [23]). Separate analysis shows that the protocol is also optimal when only manifest faults are present, and the obtained bound is $n > m$ [18]. When only symmetric faults are present, however, the protocol is definitely suboptimal, in that additional rounds can *reduce* its resilience. For example, in OMH(0) (where receivers simply accept whatever value they obtain from the transmitter), the number of symmetric-faulty receivers is irrelevant. In OMH(1), however, where receivers relay information to each other and take the majority of the values obtained, one symmetric-faulty receiver can defeat the protocol unless $n \geq 4$.

Suppose now that we use digital signatures to add authentication to OMH(r), thereby creating a protocol we can call OMHA(r). First, as Lamport, Shostak, and Pease observe [16, p.393], there is no point authenticating the final step in the protocol (i.e., the OMH(0) round), because we have point-to-point communications and the communication port on which a message arrives serves to authenticate it (this is Assumption A2); thus OMHA(0) is the same as OMH(0). For the general case, we simply modify OMH(r) so that processors sign all messages that they send, and improperly signed messages are treated by their receivers as E.

Notice that as long as authentication does not *introduce* faults (i.e., as long as a properly signed message cannot be mistakenly considered improperly signed), then OMHA(r) must have at least the fault tolerance of OMH(r), and this is independent of the cryptographic strength of the signature scheme. However, if we make the usual assumptions about the strength of the signature scheme, then authentication reduces the severity of faults that can be introduced by receivers. In particular, a symmetric-faulty receiver cannot inject a completely false value into the exchanges: at worst, it can inject an E or $R(E)$ value; similarly, an arbitrary-faulty receiver can selectively inject E and $R(E)$, or can pass on the true value that it received. (Faulty processors cannot inject $R(R(E))$ etc., because this would require an $R(E)$ correctly signed by another processor.) Unfortunately, the residual ability to inject $R(E)$ is sufficient to limit the number and combination of faults that can be tolerated by OMHA(r) to be no better, in the worst case, than for OMH(r).

This disappointing result suggests consideration of a protocol ZA(r), derived from Thambidurai and Park's protocol Z(r) in the same way that OMHA(r) is derived from OMH(r). Since Z(r) and ZA(r) lack the E, $R(E)$ distinctions of OMH(r) and OMHA(r), it follows that symmetric-faulty receivers are reduced to manifest-faulty in ZA(r). Similarly, arbitrary-faulty receivers are reduced to manifest-faulty or "nonfaulty with communications link faults," which is a case considered in Section 6. Furthermore, authentication overcomes the bug in Z(r); this bug arises in Z(1) when an arbitrary- or symmetric-faulty receiver injects spurious values into the exchanges under a manifest-faulty transmitter: the E values from the transmitter, and those relayed by good receivers, are ignored in the majority votes, which are therefore won by the spurious values injected by the faulty receiver. ZA(r) eliminates this bug because it prevents the faulty receivers manufacturing the spurious values that other proces-

sors will incorporate in their majority votes. Protocol ZA(r) is defined as follows.

ZA(0)

1. The transmitter sends its value to every receiver.
2. Each receiver uses the value received from the transmitter, or uses the value E if a missing or manifestly erroneous value is received.

ZA(r), $r > 0$

1. The transmitter signs and sends its value to every receiver.
2. For each p, let v_p be the value receiver p obtains from the transmitter, or E if no value, or a manifestly bad value, or incorrectly signed value is received.

   Each receiver p acts as the transmitter in Protocol ZA($r-1$) to communicate the value v_p to the other $n-2$ receivers.
3. For each p and q, let v_q be the value receiver p received from receiver q in step (2) (using Protocol ZA($r-1$)), or else E if no such value, or a manifestly bad value, or incorrectly signed value was received. Each receiver p calculates the majority value among all non-E values v_q received; if no such majority exists, the receiver uses some arbitrary, but functionally determined value.

We have the following results, where a, s, and m are the numbers of arbitrary-, symmetric-, and manifest-faulty processors, respectively, and n is the total number of processors.

Lemma 1 *If signatures are secure, then for any a, s, m and r, Protocol ZA(r) satisfies Validity.*

Proof: In the first round, the transmitter signs and sends its value to all receivers. Validity assumes a nonfaulty transmitter, so all nonfaulty receivers will obtain the correct value in this round. The receivers exchange values in subsequent rounds, and faulty receivers may inject faulty values into this process. However, authentication prevents the injection of any correctly signed value other than that sent by the original transmitter. Thus the only values entering the majority vote will be this value and, possibly, E. Since all good receivers obtained at least one copy of the value v directly from the transmitter, and some combination of vs and Es from other receivers, the hybrid majority will always be v. □

Theorem 2 *If signatures are secure, then for any r, Protocol ZA(r) satisfies conditions Validity and Agreement if $r \geq a$.*

Proof: The proof is by induction on r. In the base case $r = 0$ there can be no arbitrary-faulty processors, since $r \geq a$. If there are no arbitrary-faulty processors then the previous lemma ensures that ZA(0) satisfies Agreement, and Validity follows. We therefore assume that the theorem is true for ZA($r-1$) and prove it for ZA(r), $r > 0$.

First consider the case in which the transmitter is not arbitrary-faulty. Then Validity is ensured by Lemma 1, and Agreement follows from Validity. Now consider the case where the transmitter is arbitrary-faulty. There are at most a arbitrary-faulty processors, and the transmitter is one of them, so at most $a-1$ of the receivers are arbitrary-faulty. At the next stage, we have one less round to perform, and one less arbitrary fault to tolerate. Since we assume $r \geq a$, we also know $r-1 \geq a-1$, and we may therefore apply the induction hypothesis to conclude that ZA($r-1$) satisfies conditions Agreement and Validity. Hence, for each q, any two nonfaulty receivers get the same value for v_q in step (3). (This follows from Validity if one of the two receivers is processor q, and from Agreement otherwise). Hence, any two nonfaulty receivers get the same vector of values $v_1, \ldots, v_{n-1}$, and therefore obtain the same value *hybrid-majority*$(v_1, \ldots, v_{n-1})$ in step (3) (since this value is functionally determined), thereby ensuring Agreement. □

Theorem 2 shows that ZA(r) has the same (optimal) fault tolerance as SMH(r) when signatures are secure; however, ZA(r) has the significant advantage that it is not totally broken if authentication fails. In the presence of authentication failure, ZA(r) reverts to, at worst, the fault tolerance of Z(r). To be sure, Z(r) is vulnerable to certain configurations of two faults no matter how many rounds and receivers are used (that is why we developed OMH(r)), but in the important case $r = 1$, its failure mode is very precisely characterized (manifest-faulty receiver and at least one symmetric-fault or arbitrary-faulty receiver—the latter is required to break Agreement). An alternative is to use the protocol OMHA(r), whose fallback, OMH(r) is fully robust against arbitrary and manifest faults, but whose resilience in the presence of working authentication is inferior to that of ZA(r). Table 1 compares the various protocols we have discussed in terms of worst-case bounds.

Protocol	Authentication Assumptions	
	Violated	Sound
SM(r)	a = s = 0, n > m+1	n > a+s+m+1, r ≥ a
SMH(r)	a = s = 0, n > m+1	n > a+s+m+1, r ≥ a
OM(r)	n > 2a+2s+2m+r, r ≥ a	n > 2a+2s+2m+r, r ≥ a (same)
OMH(r)	n > 2a+2s+m+r, r ≥ a	n > 2a+2s+m+r, r ≥ a (same)
OMHA(r)	n > 2a+2s+m+r, r ≥ a	n > 2a+2s+m+r, r ≥ a (same)
Z(r)	n > 2a+2s+m+r, r ≥ a†	n > 2a+2s+m+r, r ≥ a† (same)
ZA(r)	n > 2a+2s+m+r, r ≥ a†	n > a+s+m+1, r ≥ a

† Z(1) also fails with a manifest-faulty transmitter and one symmetric- or arbitrary-faulty receiver; Z(r), $r > 1$, fails in additional cases.

Table 1: Comparison of Byzantine Agreement Protocols

6 Link faults

Communications failures represent an important class of faults; we call them *link* faults, with the characterization that when a nonfaulty processor sends its value v to a nonfaulty recipient over a faulty link, the value received may be either v or E.

Because they arise frequently in practice (wires and connectors are prone to noise and breakage), it is desirable to tolerate link faults efficiently. Notice that a link fault is not attributed to a processor; thus, a processor at the receiving end of a faulty link may be nonfaulty and the protocol must ensure that it satisfies the Agreement and Validity conditions. The difficulty in extending Byzantine agreement protocols to link faults is due to the fact that these faults do introduce asymmetry and are therefore as expensive to tolerate as arbitrary failures in the worst case.

We can observe that ZA(r) achieves Validity in the presence of link faults and hybrid processor faults, provided that there is path of length $r + 1$ links or less from the transmitter to each nonfaulty receiver that passes through only nonfaulty processors and good links. SMH(r) has the same bounds on Validity as ZA(r), while that of OMHA(r) is worse and difficult to characterize. We can also observe that for Agreement, a link fault is as disruptive, in the worst case, as an arbitrary fault at either the sender or receiver on the link. Thus, if link faults are attributed to either their sender or receiver, and l is the minimum number of processors needed to account for all such faults, then ZA(r) will achieve Agreement provided $r \geq a+l$. Similar worst case bounds apply for Agreement in SMH(r), while OMHA(r) requires $n > 2a + 2s + m + r + 2l$ and $r \geq a + l$.

7 Examining fault tolerance using state-exploration techniques

The worst-case bounds given above are based on rather crude ways of counting faults: there are many scenarios for the behavior of a system with, say, one arbitrary-faulty and one manifest-faulty processor and two link faults, but the worst-case analyses treat them all alike. It is therefore interesting to enquire how well the protocols perform under more fine-grained analysis and, in particular, how they perform in regions beyond those characterized by the simple worst-case bounds.

Simulation could be used to sample the behavior of the protocols, but a more attractive alternative is to use a formal state-exploration tool to examine their behavior in specific configurations under *all* scenarios. The idea is to model the system as the composition of two concurrent processes: one that injects faults and one that tolerates or diagnoses them. A state-exploration tool will then systematically explore all possible scenarios for their interaction.

We have used the Murϕ (pronounced "Murphy") system from David Dill's group at Stanford [20] for this purpose. Essentially, we provided Murϕ programs for the OMH(1), OMHA(1), Z(1), ZA(1), and SMH(1) protocols in the $n = 5$ case, and caused Murϕ to nondeterministically perform a symbolic "fault injection" (of both link faults and hybrid processor faults) and then run the protocols. By exploring all different runs (there are over 20,000 of them), Murϕ essentially undertakes exhaustive fault injection on these protocols (the process takes a couple of minutes on a SPARC 10). Of course, it would be straightforward to write a program to do this, but we consider the use of formal state-exploration tools a very promising and general

technique for the examination of algorithms for fault tolerance and diagnosis.

Our experiments confirmed the worst-case bounds on fault tolerance claimed for the various protocols in the case $n = 5$ and $r = 1$, and rediscovered the known vulnerability of Z(1) to manifest-faulty transmitters [19]. That is to say, exhaustive search of all fault configurations satisfying the bounds claimed in Table 1 for the case of $n = 5$ and $r = 1$ found no violations of Validity nor of Agreement, except for the known cases in Z(1).

However, much more interesting results were obtained when we allowed fault-injection to continue beyond the simple characterizations of worst-case fault tolerance for the protocols concerned. For example, although no five-processor, two-round protocol can withstand two link faults in the worst case, we found ZA(1) does tolerate two such faults in most cases. We therefore used our Murϕ fault-injection system to count how many scenarios caused each protocol to fail with and without the assumption of secure authentication.

Protocol	Authentication Assumptions	
	Violated	Sound
OMH(1)	25	25
OMHA(1)	25	23
Z(1)	24	24
ZA(1)	24	12
SMH(1)	43	13

Table 2: Percentage of fault configurations in a 5-plex where each protocol fails

Table 2 compares the various protocols we have discussed, using exhaustive state exploration to calculate the percentage of fault configurations that caused the protocols to fail. Overall, it seems that ZA(1) is the most resilient of these protocols under the combination of hybrid and link faults, though more experiments are needed to confirm this.

Fault configurations consist of an assignment of fault class (good, manifest, symmetric, or arbitrary) to each processor, and an assignment of up to three faulty links between processors. We excluded configurations with link faults emanating from arbitrary or manifestly faulty transmitters, or arriving at faulty receivers (such link faults have no real impact on system behavior). For each configuration, we tested whether any scenario of messages by the faulty processors could cause good receivers to disagree or cause a good receiver to fail to agree with the transmitter. For each protocol, we then calculated the percentage of all fault configurations for which such failure was possible.

The newest release of the Murϕ system automatically detects and exploits symmetry in appropriately written specifications, reducing the search space dramatically. For example, the configuration where all processors are good except that the third receiver is manifest-faulty is isomorphic to the case when all processors are good except the second receiver, and Murϕ only explores one of these alternatives. Symmetries are used in the assignment of faulty links as well as in the assignment of behaviors to processors. Because of these symmetry reductions, not all configurations are counted individually, so the numbers in Table 2 should be taken to indicate relative, not absolute, performance. We further reduced the set of configurations to require at least one good receiver, since otherwise validity and agreement are trivially satisfied. We excluded symmetric-faulty processors sending manifestly bad (E) values, since this would amount to the same thing as a manifest fault, and we also excluded the case of a symmetric-faulty transmitter since there is very little difference between this case and that when the transmitter is good. However, we did allow an arbitrary-faulty transmitter to behave in any way, including the possibility of behaving as good, symmetric- or manifest-faulty, as well as sending various combinations of good, wrong, and E values.

For the authenticated protocols, faulty receivers were not allowed to send data values other than that received from the transmitter. Thus for algorithm ZA(1) arbitrary-faulty receivers are only able to send manifestly bad (E) values or the correct value. In algorithm OMHA(1), arbitrary-faulty receivers also have the opportunity to send $R(E)$ and, as discussed earlier, this is the main source of brittleness of OMHA(1). We further make the assumption in these experiments that authentication never leads to good processors discarding good messages. These factors, taken together, significantly reduce the total number of configurations that need to be considered, but do not effect the relative numbers of configurations where the various algorithms behave acceptably.

The table shows that the authenticated protocol ZA(1) wrings the maximum fault tolerance from a given amount of redundant hardware, and outperforms the classical Signed Messages protocol whether or not signatures are secure (dramatically so if signatures are insecure). ZA(1) is also superior in overall resilience to OMHA(1). This is not to say that ZA(1) is uniformly superior to OMHA(1). Consider a good

transmitter with link faults to all receivers except p, and p has a link fault to receiver q. Under ZA(1), q decides on E and all the other receivers decide on the value sent by the transmitter to p, thereby violating Agreement. Under OMHA(1) all receivers settle on E.

Note that we are testing the fault tolerance of these protocols well beyond their usually claimed fault tolerance: only approximately five percent of all fault configurations we studied fall within the worst-case bounds of the protocols. Thus, all these protocols are far more tolerant of faults than their simple worst-case bounds would suggest.

8 Conclusion

The assumptions required of the authentication mechanism in Byzantine agreement protocols that use "signed messages" are stronger than generally realized, and require that digital signatures are used with great care. Violation of these assumptions can cause the protocols to fail. We have presented new protocols that combine authentication with "oral messages" protocols so that additional resilience is obtained when the authentication assumptions are sound, but the resilience of the unauthenticated protocol is retained when authentication assumptions are violated.

When the authentication assumptions are sound, one of these new protocols, called ZA(r), matches the fault tolerance of the classical signed messages protocol under a hybrid fault model, and surpasses it when communications link faults are considered. ZA(r) also performs well overall when authentication assumptions are violated, but has an unfortunate "hole" in its worst-case bound (it is vulnerable when the transmitter is manifest-faulty). Another of the new protocols, OMHA(r) may be preferred if this case is considered important, though it is less resilient to link faults than ZA(r).

These new protocols are superior to other known protocols in properties and measures of practical interest, and we recommend them for general use. They are particularly attractive in security-critical systems where authentication may be subjected to sophisticated cryptographic attack, and in safety-critical embedded systems where maximum resilience is required but where only short or cryptographically weak signatures (e.g., checksums) may be feasible. Selection of the most suitable protocol for a given system must obviously depend on the expected modes and frequencies of faults, and the consequences of system failure.

Our use of the state-exploration system Murϕ to perform symbolic "fault injection" is, we believe, novel. It suggests a very promising new application area for this class of formal methods tools, and one that we intend to pursue in future work.

Acknowledgments

Our understanding of these topics has benefitted greatly from discussions with Chris Walter and Michele Hugue (both then with Allied Signal). Comments by the anonymous reviewers were also very helpful. Malte Borcherding of the University of Karlsruhe pointed out some errors in the original paper.

References

Papers by SRI authors can generally be retrieved from `http://www.csl.sri.com/fm.html`.

[1] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 122–136, Oakland, CA, May 1994. IEEE Computer Society.

[2] Birgit Baum-Waidner. Byzantine agreement with a minimum number of messages both in the faultless and worst case. In *Fault Tolerant Computing Symposium 23* [14], pages 554–563.

[3] Malte Borcherding. Efficient failure discovery with limited authentication. In *15th International Conference on Distributed Computing Systems*, pages 78–82, Vancouver, Canada, May 1995. IEEE Computer Society.

[4] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–650, November 1976.

[5] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, November 1983.

[6] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM*, 32(1):191–204, January 1985.

[7] Danny Dolev, Rüdiger Reischuk, and H. Raymond Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, October 1990.

[8] Klaus Echtle. Fault masking with reduced redundant communication. In *Fault Tolerant Computing Symposium 16*, pages 178–183, Vienna, Austria, July 1986. IEEE Computer Society.

[9] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, July 1985.

[10] Paul D. Ezhilchelvan. Early stopping algorithms for distributed agreement under fail-stop, omission, and timing fault types. In *6th Symposium on Reliability in Distributed Software and Database Systems*, pages 201–212, Williamsburg, VA, March 1987. IEEE Computer Society.

[11] M. Fischer and N. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14:183–186, 1982.

[12] F. Di Giandomenico, M. L. Guidotti, F. Grandoni, and L. Simoncini. A graceful dependable algorithm for Byzantine agreement. In *6th Symposium on Reliability in Distributed Software and Database Systems*, pages 188–200, Williamsburg, VA, March 1987. IEEE Computer Society.

[13] L. Gong. Variations on the themes of message freshness and replay. In *Proceedings of the Computer Security Foundations Workshop VII*, pages 131–136, Franconia, NH, June 1993. IEEE Computer Society.

[14] *Fault Tolerant Computing Symposium 23*, Toulouse, France, June 1993. IEEE Computer Society.

[15] R. M. Kieckhafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37(4):398–405, April 1988.

[16] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[17] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 292–304, Elounda, Greece, June/July 1993. Springer-Verlag.

[18] Patrick Lincoln and John Rushby. Formal verification of an algorithm for interactive consistency under a hybrid fault model. Technical Report SRI-CSL-93-2, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1993. Also available as NASA Contractor Report 4527, July 1993.

[19] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23* [14], pages 402–411.

[20] Ralph Melton and David L. Dill. *Murϕ Annotated Reference Manual.* Computer Science Department, Stanford University, Stanford, CA, March 1993.

[21] Judy H. Moore. Protocol failures in cryptosystems. *Proceedings of the IEEE*, 76(5):594–602, May 1988.

[22] National Institute of Standards and Technology. The digital signature standard. *Communications of the ACM*, 37(7):36–40, July 1992.

[23] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[24] Michael Reiter. A secure group membership protocol. In *Proceedings of the Symposium on Research in Security and Privacy*, pages 176–189, Oakland, CA, May 1994. IEEE Computer Society.

[25] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[26] John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed Computing*, pages 304–313, Los Angeles, CA, August 1994. Association for Computing Machinery.

[27] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[28] T.K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

[29] Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, Columbus, OH, October 1988. IEEE Computer Society.

[30] C. J. Walter, N. Suri, and M. M. Hugue. Continual online diagnosis of hybrid faults. In F. Cristian, G. Le Lann, and T. Lunt, editors, *Dependable Computing for Critical Applications—4*, volume 9 of *Dependable Computing and Fault-Tolerant Systems*, pages 233–249. Springer-Verlag, Vienna, Austria, January 1994.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms

John Rushby

Abstract—**Many critical real-time applications are implemented as time-triggered systems. We present a systematic way to derive such time-triggered implementations from algorithms specified as functional programs (in which form their correctness and fault-tolerance properties can be formally and mechanically verified with relative ease). The functional program is first transformed into an untimed synchronous system, and then to its time-triggered implementation. The first step is specific to the algorithm concerned, but the second is generic and we prove its correctness. This proof has been formalized and mechanically checked with the PVS verification system. The approach provides a methodology that can ease the formal specification and assurance of critical fault-tolerant systems.**

Keywords—**Formal methods, formal verification, time-triggered algorithms, synchronous systems, PVS.**

I. Introduction

SYNCHRONOUS systems are distributed computer systems where there are known upper bounds on the time that it takes nonfaulty processors to perform certain operations, and on the time that it takes for a message sent by one nonfaulty processor to be received by another. The existence of these bounds simplifies the development of fault-tolerant systems because nonfaulty processes executing a common algorithm can use the passage of time to predict each others' progress. This property contrasts with asynchronous systems, where there are no upper bounds on processing and message delays, and where it is therefore provably impossible to achieve certain forms of consistent knowledge or coordinated action in the presence of even simple faults [1, 2].

For these reasons, fault-tolerant systems for critical control applications in aircraft, trains, automobiles, and industrial plants are usually based on the synchronous approach, though they differ in the extent to which the basic mechanisms of the system really do guarantee satisfaction of the synchrony assumption.

With systems based on conventional "commercial off the shelf" (COTS) components, synchrony is merely an assumption—these systems employ scheduling algorithms that can miss deadlines, their operating systems admit the possibility of buffer overflows, they use contention buses such as Ethernet, and they have other characteristics that allow occasional violations of claimed time bounds. Violation of the synchrony assumption may lead to failure of the higher-level system components that depend on it, so adopting this assumption when it is only probabilistically valid has ramifications on overall system reliability. Notice that adding timeouts does not make an asynchronous system synchronous [3].

While probabilistic satisfaction of the synchrony assumption may be "good enough" for less critical applications, those that are truly critical must either rest on weaker assumptions, or must be specially constructed to ensure that the assumption is unconditionally valid. Those that take the latter course often build on mechanisms that are not merely synchronous, but synchronized and time-triggered: the clocks of the different processors are kept close together, processors perform their actions at specific times, and tasks and messages are globally and statically scheduled. The buses and operating systems used in these contexts are specialized and dedicated to satisfaction of the synchrony hypothesis [4]. The Honeywell SAFEbus™ [5,6] that provides the safety-critical backplane for the Boeing 777 Airplane Information Management System (AIMS) [7, 8], the control system for the Shinkansen (Japanese Bullet Train) [9], and the Time-Triggered Protocol (TTP) for safety-critical automobile functions [10] all use this approach.

A number of basic functions have been identified that provide important building blocks in the construction of fault-tolerant synchronous systems [11, 12]; these include consensus (also known as interactive consistency and Byzantine agreement) [13], reliable and atomic broadcast [14], and group membership [15]. Numerous algorithms have been developed to perform these functions and, because of their criticality and subtlety, several of them have been subjected to detailed formal [16–18] and mechanically checked [19–23] verifications, as have their combination into larger functions such as diagnosis [24], and their synthesis into a fault-tolerant architecture based on active (state-machine) replication [25, 26].

Formal, and especially mechanically-checked, verification of these algorithms is still something of a *tour de force*, however. To have real impact on practice, we need to reduce the difficulty of formal verification in this domain to a routine and largely automated process. In order to achieve this, we should study the sources of difficulty in existing treatments and attempt to reduce or eliminate them. In particular, we should look for opportunities for systematic treatments: these may allow aspects common to a range of algorithms to be treated in a uniform way, and may even

The author is with the Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA. Email: Rushby@csl.sri.com

This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044, by the National Science Foundation under contract CCR-9509931, and by NASA Langley Research Center under contract NAS1-20334.

allow some of those aspects to be broken out and verified in a generic manner once and for all.

There is a wide range in the apparent level of difficulty and detail in the mechanized verifications cited above. Some of the differences can be attributed to the ways in which the problems are formalized or to the different resources of the formal specification languages and theorem provers employed. For example, Rushby [19] and Bevier and Young [23] describe mechanically checked formal verifications of the same "Oral Messages" algorithm [27] for the consensus problem that were performed using different verification systems. Young [28] argues that differences in the difficulty of these treatments (that of [19] is generally considered simpler and clearer than that of [23]) are due to choices in the way things are formalized, and not to the capabilities of the tools employed. We may assume that such differences will be reduced or eliminated as experience is gained and the better choices become more widely known.

More significant than differences due to *how* things are formalized are differences due to *what* is formalized, and the level of detail considered necessary. For example, both verifications of the Oral Messages algorithm mentioned above specify the algorithm as a functional program and the proofs are conventional inductions. Following this approach, the special case of a two-round algorithm (a variant of the algorithm known as OM(1)) is specified in [22] in a couple of lines and its verification is almost completely automatic. In contrast, the treatment of OM(1) in [18] is long and detailed and quite complicated. The reason for its length and complexity is that this treatment explicitly considers the distributed, message passing character of the intended implementation, and calculates tight real-time bounds on the timeouts employed. All these details are abstracted away in the treatments using functional programs—but this does not mean these verifications are inferior to the more detailed analyses: on the contrary, I would argue that they capture the essence of the algorithms concerned (i.e., they explain *why* the algorithm is fault tolerant) and that message-passing and real-time bounds are implementation details that ought to be handled separately. In fact, most of the papers that introduce the algorithms concerned, and the standard textbook [29], use a similarly abstract and time-free treatment. On the other hand, it is undeniably important also to verify a specification that is reasonably close to the intended implementation, and to establish that the correct timeouts are used, and that the concrete fault modes match those assumed in the more abstract treatment.

The natural resolution for these competing claims for abstractness and concreteness is a hierarchical approach in which the essence of the algorithm is verified in an abstract formulation, and a more realistic formulation is then shown to be a refinement, in some suitable sense, of the abstract formulation. This may not always be possible (e.g., for event-based systems) but, when it is, we may hope that the refinement argument will be a routine calculation of timeouts and other concrete details.

The purpose of this paper is to present such a hierarchical treatment for the important case of time-triggered implementations of round-based algorithms, and to show that most of the details of refinement to a concrete formulation can be worked out once and for all.

II. Round-Based Algorithms

In her textbook [29], Nancy Lynch identifies algorithms for the synchronous system model with those that execute in a series of "rounds." Rounds have two phases: in the first, each processor[1] sends a message to some or all of the other processors (different messages may be sent to different processors; the messages depend on the current state of the sending processor); in the second phase, each processor changes its state in a manner that depends on its current state and the collection of messages it received in the first phase. There is no notion of real-time in this model: messages are transferred "instantaneously" from senders to recipients between the two phases. The processors operate in lockstep: all of them perform the two phases of the current round, then move on to the first phase of the next round, and so on.

Several of the algorithms of interest here were explicitly formulated in terms of rounds when first presented, and others can easily be recast into this form. For example, the Oral Messages algorithm for consensus, OM(1), requires two rounds as follows.

Algorithm OM(1).

Round 0:

Communication Phase: A distinguished processor called the *transmitter* sends a value to all the other processors, which are called *receivers*; the receivers send no messages.

Computation Phase: Each receiver stores the value received from the transmitter in its state.

Round 1:

Communication Phase: Each receiver sends the value it received from the transmitter to all the other receivers; the transmitter sends no message.

Computation Phase: Each receiver sets the "decision" component of its state to the majority value among those received from the other receivers and that (stored in its state) received from the transmitter.

In the presence of one or fewer arbitrary faults, OM(1) ensures that all nonfaulty receivers decide on the same value and, if the transmitter is nonfaulty, that value is the one sent by the transmitter.

There are two different ways to implement round-based algorithms. In the *time-triggered* approach, the implementation is very close to the model: the processors are closely synchronized (e.g., to within a couple of bit-times in the case of SAFEbus) and all run a common, deterministic

[1] I refer to the participants as processors to stress that they are assumed to fail independently; the agents that perform these actions will actually be processes.

schedule that will cause them to execute specific algorithms at specific times (according to their local clocks). The sequencing of phases and rounds is similarly driven by the local clocks, and communication bandwidth is also allocated as dedicated, fixed, time slots. The first (communication) phase in each round must be sufficiently long that all nonfaulty processors will be able to exchange messages successfully; consequently, no explicit timeouts are needed: a message that has not arrived by the time the second (computation) phase of a round begins is implicitly timed out.

Whereas the allocation of resources is statically determined in the time-triggered approach, in the other, *event-triggered*, approach, resources are scheduled dynamically and processors respond to events as they occur. In this implementation style, the initiation of a protocol may be triggered by a local clock, but subsequent phases and rounds are driven by the arrival of messages. In Lamport and Merz' treatment of OM(1), for example, a receiver that has received a message from the transmitter may forward it immediately to the other receivers without waiting for its clock to indicate that the next round has started (in other words, the pacing of phases and rounds is determined locally by the availability of messages). Unlike the time-triggered approach, messages may have to be explicitly timed out in the event-triggered approach. For example, in Lamport and Merz' treatment of OM(1), a receiver will not wait for relayed messages from other receivers beyond $2\delta + \epsilon$ past the start of the algorithm (where δ is the maximum communication delay and ϵ the maximum time that it can take a receiver to decide to relay a message).

Event-triggered systems are generally easier to construct than time-triggered ones (which require a big planning and scheduling effort upfront) and achieve better CPU utilization under light load. On the other hand, Kopetz [4,10,30] argues persuasively that time-triggered systems are more predictable (and hence easier to verify), easier to test, easier to compose together, make better use of broadcast communications bandwidth, can operate closer to capacity, and are generally to be preferred for truly critical applications. The previously mentioned SAFEbus for the Boeing 777, the Shinkansen train control system, and the TTP protocol for automobiles are all time-triggered.

Our goal is a systematic method for transforming round-based protocols from very abstract functional programs, whose properties are comparatively easy to formally and mechanically verify, down to time-triggered implementations with appropriate timing constraints and consideration for realistic fault modes. The transformation is accomplished in two steps: first from a functional program to an (untimed) synchronous system, then to a time-triggered implementation. The first step is systematic but must be undertaken separately for each algorithm (see Section IV); the other is generic and deals with a large class of algorithms and fault assumptions in a single verification. This generic treatment of the second step is described in the following section.

III. Implementation of Round-Based Algorithms as Time-Triggered Systems

The issues in transforming an untimed round-based algorithm to a time-triggered implementation are basically to ensure that the timing and duration of events in the communication phase are such that messages between nonfaulty processors always arrive in the communication phase of the same round, and fault modes are interpreted appropriately. To verify the transformation, we introduce formal models for untimed synchronous systems and for time-triggered systems, and then establish a simulation relation between them. We verify the simulation by means of a traditional mathematical proof, and then describe a mechanized verification performed using the PVS verification system [31].

A. Synchronous Systems

For the untimed case, we use Nancy Lynch's formal model for synchronous systems [29, Chapter 2], with some slight adjustments to the notation that make it easier to match up with the mechanically verified treatment.

Definition 1: Untimed Synchronous Systems.

We assume a set *mess* of messages that includes a distinguished value *null*, and a set *proc* of processors. Processors are partially connected by directed *channels*; each channel can be thought of a buffer that can hold a single message. Associated with each processor p are the following sets and functions.

- A set of processors $out\text{-}nbrs_p$ to which p is connected by outgoing channels.
- A set of processors $in\text{-}nbrs_p$ to which p is connected by incoming channels; the function $inputs_p : in\text{-}nbrs_p \rightarrow mess$ gives the message contained in each of those channels.
- A set $states_p$ of states with a nonempty subset $init_p$ of initial states. It is convenient to assume that there is a component in the state that counts rounds; this counter is zero in initial states.
- A function $msg_p : states_p \times out\text{-}nbrs_p \rightarrow mess$ that determines the message to be placed in each outgoing channel in a way that depends on the current state.
- A function $trans_p : states_p \times inputs_p \rightarrow states_p$ that determines the next state, in a way that depends on the current state and the messages received in the incoming channels.

The system starts with each processor in an initial state. All processors p then repeatedly perform the following two actions in lockstep.

Communication Phase: apply the message generation function msg_p to the current state to determine the messages to be placed in each outgoing channel. (The message value *null* is used to indicate "no message.")

Computation Phase: apply the state transition function $trans_p$ to the current state and the message held in each

incoming channel to yield the next state (with the round counter incremented).

□

A particular algorithm is specified by supplying interpretations to the various sets and functions identified above.

A.1 Faults

Distributed algorithms are usually required to operate in the presence of faults: the specific kinds and numbers of faults that may arise constitute the *fault hypothesis*. Usually, processor faults are distinguished from communication faults; the former can be modeled by perturbations to the transition functions $trans_p$, and the latter by allowing the messages received along a channel to be changed from those sent. Following [29, page 20], an *execution* of the system is then an infinite sequence of triples

$$(S_0, M_0, N_0), (S_1, M_1, N_1), (S_2, M_2, N_2), \ldots$$

where S_r is the global state at the start of round r, M_r is the collection of messages placed in the communication channels, and N_r is the (possibly different) collection of messages received.

Because our goal is to show that a time-triggered implementation achieves the same behavior as the untimed synchronous system that serves as its specification, we will need some way to ensure that faults match up across the two systems. For this reason, I prefer to model processor and communication faults by perturbations to the $trans_p$ and msg_p functions, respectively (rather than allowing messages received to differ from those sent); no faulty behaviors are lost by this change. In particular, I assume that the current round number is recorded as part of the state and that if processor p is faulty in round r, with current state s and the values of its input channels represented by the array i, then $trans_p(s, i)$ may yield a value other than that intended; similarly, if the channel from p to q is faulty, then the value $msg_p(s)(q)$ may be different than intended (and may be *null*). Exactly how these values may differ from those intended depends on the fault assumption. For example, a crash fault in round r results in $trans_p(s, i) = s$ and $msg(s)(q) = null$ for all i, q, and states s whose round component is r or greater. Notice that although $trans_p$ and msg_p may no longer be the intended functions, they are still functions; in fact, there is no need to suppose that the $trans_p$ and msg_p were *changed* when the fault arrived in round r: since the round counter is part of the state, we can just assume these functions behave differently than intended when applied to states having round counters equal or greater than r.

The benefit of this treatment is that, since $trans_p$ and msg_p are uninterpreted, they can represent any algorithm and any fault behavior whatsoever; if we can show that a time-triggered system supplied with arbitrary $trans_p$ and msg_p functions has the same behavior as the untimed synchronous system supplied with the same functions, then this demonstration encompasses behavior in the presence of faults as well as the fault-free case. Furthermore, since we no longer need to hypothesize that faults can cause differences between those messages sent and those received (we instead assume the fault is in msg_p and the "different" messages were actually sent), executions can be simplified from sequences of triples to simple sequences of states

$$S_0, S_1, S_2, \ldots$$

where S_r is the global state at the start of round r. Consequently, to demonstrate that a time-triggered system implements the behavior specified by an untimed synchronous system, we simply need to establish that both systems have the same execution sequences; by mathematical induction, this will reduce to showing that the global states of the two systems are the same at the start of each round r.

B. Time-Triggered Systems

For the time-triggered system, we elaborate the model of the previous section as follows.

Each processor is supplied with a clock that provides a reasonably accurate approximation to "real" time. Following [32], we distinguish two notions of time: *clocktime*, denoted C is the local notion of time supplied by each processor's clock, while *realtime*, denoted $\mathcal{R}$ is an abstract global quantity. We follow the usual convention and denote clocktime quantities by upper case Roman or Greek letters, and realtime quantities by lower case letters.

Formally, processor p's clock is a function $C_p : \mathcal{R} \to \mathcal{C}$. The intended interpretation is that $C_p(t)$ is the value of p's clock at realtime t.[2] The clocks of nonfaulty processors are assumed to be well-behaved in the sense that they satisfy the following assumptions.

Assumption 1: Monotonicity. Nonfaulty clocks are monotonic increasing functions:

$$t_1 < t_2 \supset C_p(t_1) < C_p(t_2).^3$$

Satisfying this assumption requires some care in implementation, because clock synchronization algorithms can make adjustments to clocks that cause them to jump backwards. Lamport and Melliar-Smith describe some solutions [32], and a particularly clever and economical technique for one particular algorithm is introduced by Torres-Pomales [33] and formally verified by Miner and Johnson [34]. Schmuck and Cristian [35] examine the general case and show that monotonicity can be achieved with no loss of precision.

Assumption 2: Clock Drift Rate. Nonfaulty clocks drift from realtime at a rate bounded by a small positive quantity ρ:

$$(1 - \rho)(t_1 - t_2) \leq C_p(t_1) - C_p(t_2) \leq (1 + \rho)(t_1 - t_2).$$

This assumption concerns the hardware clocks employed Inexpensive devices can achieve $\rho < 10^{-6}$.

[2] In the terminology of [32], these are actually "inverse" clocks.
[3] The symbol $\supset$ indicates logical implication.

Assumption 3: Clock Synchronization. The clocks of nonfaulty processors are synchronized within some small clocktime bound Σ:

$$|C_p(t) - C_q(t)| \leq \Sigma.$$

This assumption can be discharged by a suitable clock synchronization algorithm. There are many such algorithms, several of which have been formally verified [36–41].

Definition 2: Time-Triggered Systems.

The feature that characterizes a time-triggered system is that all activity is driven by a global schedule: a processor performs an action when the time on its local clock matches that for which the action is scheduled. In our formal model, the schedule is a function $sched : \mathbb{N} \rightarrow C$, where $sched(r)$ is the clocktime at which round r should begin. The duration of the r'th round is given by

$$dur(r) = sched(r + 1) - sched(r).$$

In addition, there are fixed global clocktime constants D and P that give the offsets into each round when messages are sent, and when the computation phase begins, respectively. Obviously, we need the following constraint.

Constraint 1: $0 < D < P < dur(r)$.

Notice that the duration of the communication phase is fixed (by P); it is only the duration of the computation phase that can differ from one round to another.[4]

The states, messages, and channels of a time-triggered system are the same as those for the corresponding untimed synchronous system, as are the transition and message functions. In addition, processors have a one-place buffer for each incoming message channel.

The time-triggered system operates as follows. Initially each processor is in an initial state, with its round counter zero and its clock synchronized with the others and initialized so that $C_p(t_0) \leq sched(0)$, where t_0 is the current realtime. All processors p then repeatedly perform the following two actions.

Communication Phase: This begins when the local clock reads $sched(r)$, where r is the current value of the round counter. Apply the message generation function msg_p to the current state to determine the messages to be sent on each outgoing channel. The messages are placed in the channels at local clock time $sched(r) + D$. Incoming messages that arrive during the communication phase (i.e., no later than $sched(r) + P$) are moved to the corresponding input buffer where they remain stable through the computation phase. These buffers are initialized to *null* at the beginning of each communication phase and their value is unspecified if more than one message arrives on their associated communications channel in a given communication phase.

Computation Phase: This begins at local clock time $sched(r) + P$. Apply the state transition function $trans_p$ to the current state and the messages held in the input buffers to yield the next state. The computation will be complete at some local clock time earlier than $sched(r + 1)$. Increment the round counter, and wait for the start of the next round.

□

Message transmission in the communication phase is explained as follows. We use $sent(p, q, m, t)$ to indicate that processor p sent message m to processor q (a member of $out\text{-}nbrs(p)$) at real time t (which must satisfy $C_p(t) = sched(r) + D$ for some round r). We use $recv(q, p, m, t)$ to indicate that processor q received message m from processor p (a member of $in\text{-}nbrs(q)$) at real time t (which must satisfy the constraint $sched(r) \leq C_q(t) < sched(r) + P$ for some round r). These two events are related as follows.

Assumption 4: Maximum Delay. When p and q are nonfaulty processors,

$$sent(p, q, m, t) \supset recv(q, p, m, t + d)$$

for some $0 \leq d \leq \delta$.

In addition, we require no spontaneous generation of messages (i.e., $recv(q, p, m, t)$ only if there is a corresponding $sent(p, q, m, t')$ with $t' < t$).

Provided there is exactly one $recv(q, p, m, t)$ event for each p in the communication phase for round r on processor q (as there will be if p is nonfaulty), that unique message m is moved into the input buffer associated with p on processor q before the start of the computation phase for that round and remains there throughout the phase.

Because the clocks are not perfectly synchronized, it is possible for a message sent by a processor with a fast clock to arrive while its recipient is still on the previous round. It is for this reason that we do not send messages until D clocktime units into the start of the round. In general, we need to ensure that a message from a processor in round r cannot arrive at its destination before that processor has started round r, nor after it has finished the communication phase for round r. We must establish constraints on parameters to ensure these conditions are satisfied.

Now processor p sends its message to processor q, say, at realtime t where $C_p(t) = sched(r) + D$ and, by the maximum delay assumption, the message will arrive at realtime $t + d$ where $d \leq \delta$. We need to be sure that

$$sched(r) \leq C_q(t + d) < sched(r) + P. \quad (1)$$

By clock synchronization, we have $|C_q(t) - C_p(t)| \leq \Sigma$; substituting $C_p(t) = sched(r) + D$ we obtain

$$-\Sigma \leq C_q(t) - sched(r) - D \leq \Sigma. \quad (2)$$

[4]There is no difficulty in generalizing the treatment to allow the time at which messages are sent, and the duration of the communication phase, to vary from round to round. That is, the fixed clocktime constants D and P can be systematically replaced by functions $D(r)$ and $P(r)$, respectively. This generalization was developed during the mechanized verification; see Section III-D.

By the monotonic clocks assumption, this gives

$$sched(r) + D - \Sigma \leq C_q(t) \leq C_q(t + d)$$

and so the first inequality in (1) can be ensured by

Constraint 2: $D \geq \Sigma$.

The clock synchronization calculation (2) above also gives

$$C_q(t) \leq sched(r) + D + \Sigma$$

and the clock drift rate assumption gives

$$(1 - \rho)d \leq C_q(t + d) - C_q(t) \leq (1 + \rho)d$$

from which it follows that

$$C_q(t + d) \leq C_q(t) + (1 + \rho)d.$$

Combining these and recalling that $d \leq \delta$, the second inequality in (1) can be ensured by

Constraint 3: $P > D + \Sigma + (1 + \rho)\delta$.

B.1 Faults

We will prove that a time-triggered system satisfying the various assumptions and constraints identified above achieves the same behavior as an untimed synchronous system supplied with the same $trans_p$ and msg_p functions. I explained earlier that faults are assumed to be modeled in the $trans_p$ and msg_p functions; by using the same functions in both the untimed and time-triggered systems, we ensure that the latter inherits the same fault behavior and any fault-tolerance properties of the former. Thus, if we have an algorithm that has been shown, in its untimed formulation, to achieve some fault-tolerance properties (e.g., "this algorithm resists a single Byzantine fault or two crash faults"), then we may conclude that the implementation has the same properties.

This simple view is somewhat compromised, however, because the time-triggered system contains a mechanism—time triggering—that is not present in the untimed system. This mechanism admits faults (notably, loss of clock synchronization) that do not arise in the untimed system.

The implementation of a time-triggered system is required to satisfy the synchrony hypothesis and the four assumptions about nonfaulty clocks listed previously. These can be achieved using a suitable fault-tolerant clock synchronization algorithm. The algorithm and its various parameters must be chosen to tolerate the number and kinds of faults specified for the system concerned. For example, the clock synchronization algorithm of TTP (which is based on that of [42]) has recently been formally verified using PVS, and shown to satisfy the assumptions we require [41]. However, a clock synchronization algorithm only constrains the behavior of nonfaulty clocks: a processor with a faulty clock may behave in a way that violates the fault model of our time-triggered construction. For example, if one processor's clock drifts to such an extent that it is in the wrong round, then it will execute the transition and message functions appropriate to that round and will supply systematically incorrect messages to the other processors. This could appear as Byzantine behavior at the level of the untimed synchronous algorithm. Less drastic synchronization faults may leave a processor in the right round, but sending messages at the wrong time, so that they arrive during the computation phases of other (correct) processors, possibly disrupting their activity.

The implementation of the time-triggered system must include mechanisms that transform faults (such as those due to loss of clock synchronization) that are outside the model considered here, into those that are adequately modeled as perturbations to the $trans_p$ and msg_p functions. For example, the round number should be included in messages, so that those from the wrong round can be rejected at the message communication layer (thereby reducing the manifestation of such a synchronization fault to failsilence). TTP goes further and includes all critical state information (operating mode, time, and group membership) in its messages as part of the CRC calculation [10]; messages from a processor that is out of step with respect to any of these items will be rejected by the TTP controllers of other processors.

The impact of messages that arrive in the right round but at the wrong time can be partly countered by moving messages from their input channels to an input buffer at the start of the communication phase: this shields the receiving processor from any changes in channel contents during the computation phase. However, the performance of the computation phase may be degraded by the need to handle interrupts from messages arriving unexpectedly, thereby challenging the synchrony hypothesis. Strong elimination of such timing faults is achieved in practice by techniques to control the "babbling idiot" fault mode. This fault mode occurs when a faulty or unsynchronized processor transmits at arbitrarily wrong times. As well as undesirable manifestations at the synchronous system level, this fault is potentially devastating to the underlying implementation if that implementation multiplexes its communication channels onto shared buses—because the faulty processor can then disrupt the communications of nonfaulty processors. Babbling is eliminated by use of a Bus Interface Unit (BIU) that only grants its processor access to the bus at appropriate times. For example, in SAFEbus, processors are paired, with each member of a pair controlling the other's BIU; in TTP, the BIU has an independent clock and independent knowledge of the schedule [43]. In both cases, babbling can occur only if there are undetected double failures. These mechanisms prevent messages being sent at inappropriate times and ensure that the fault modes of the time-triggered implementation correspond those assumed for the untimed synchronous system.

C. Verification

We now need to show that a time-triggered system achieves the same behavior as its corresponding untimed synchronous system. We do this in the traditional way by

establishing a simulation relationship between the states of an execution of the time-triggered system and those of the corresponding untimed execution. It is usually necessary to invent an "abstraction function" to relate the states of an implementation to those of its specification; here, however, the states of the two systems are the same, and the only difficult point is to select the moments in time at which states of the time-triggered system should correspond to those of the untimed system.

The untimed system makes progress in discrete global steps: all component processors perform their communication and computation phases in lockstep, so it is possible to speak of the complete system being in a round r. The processors of the time-triggered system, however, progress separately at a rate governed by their internal clocks, which are imperfectly synchronized, so that one processor may still be on round r while another has moved on to round $r+1$. We need to establish some consistent "cut" through the time-triggered system that provides a global state in which all processors are at the same point in the same round. In some treatments of distributed systems, it is not necessary for the global cut to correspond to a snapshot of the system at a particular realtime instant: the cut may be an abstract construction that has no direct realization. In our case, however, it is natural to assume that the time-triggered system is used in some control application and that outputs of the individual processors (i.e., some functions of their states) are used to provide redundant control signals in real time—for example, a typical application will be one in which the outputs of the processors are subjected to majority voting, or separately drive some actuator in a "force-summing" configuration.[5] Consequently, we do want to identify the cut through the system with its global state at a specific real time instant.

In particular, we need some realtime instant $gs(r)$ that corresponds to the "global start" of the r'th round. We want this instant to be one in which all nonfaulty processors have started the r'th round, but have not yet started its computation phase (when they will change their states).

We can achieve this by defining the global start time $gs(r)$ for round r to be the realtime when the processor with the slowest clock begins round r. That is, $gs(r)$ satisfies the following conditions:

$$\forall q : C_q(gs(r)) \geq sched(r), \tag{3}$$

and

$$\exists p : C_p(gs(r)) = sched(r) \tag{4}$$

(intuitively, p is the processor with the slowest clock).

Since the processors are not perfectly synchronized, we need to be sure that they cannot drift so far apart that some processor q has already reached its computation phase—or is even on the next round—at $gs(r)$. Thus, we need

$$\forall q : C_q(gs(r)) < sched(r) + P. \tag{5}$$

[5]For example, the outputs of different processors may energize separate coils of a single solenoid, or multiple hydraulic pistons may be linked to a single shaft (see, e.g., [44, Figure 3.2–2]).

By (3) we have $C_q(gs(r)) = sched(r) + X$ for some $X \geq 0$, and (4) plus the clock synchronization assumption then gives $X \leq \Sigma$. Now processor q will still be on round r and in its communication phase provided $X < P$ and this is ensured by the inequality just derived when taken together with Constraint 3.

We now wish to establish that the global state of a time-triggered system at time $gs(r)$ will be the same as that of the corresponding untimed synchronous system at the start of its r'th round. We denote the global state of the untimed system at the start of the r'th round by $gu(r)$ (for *global untimed*). Global states are simply arrays of the states of the individual processors, so that the state of processor p at this point is $gu(r)(p)$. Similarly, the global state of the time-triggered system at time $gs(r)$ is denoted $gt(r)$ (for *global timed*), and the state of its processor p is $gt(r)(p)$. We can now state and prove the desired result.

Theorem 1: Given the same initial states, the global states of the untimed and time-triggered systems are the same at the beginning of each round:

$$\forall r : gt(r) = gu(r).$$

Proof: The proof is by induction.

Base case. This is the case $r = 0$. Both systems are then in their initial states which, by hypothesis, are the same.
Inductive step. We assume the result for r and prove it for $r+1$. For the untimed case, the message $inputs_q(p)$ from processor p received by q in the r'th round is $msg_p(gu(r)(p))(q)$.[6]
By the inductive hypothesis, the global state of processor p in the time-triggered system at time $gs(r)$ is $gu(r)(p)$ also. Furthermore, processor p is in its communication phase (ensured by (5)) and has not changed its state since starting the round. Thus, at local clocktime $sched(r) + D$, it sends $msg_p(gu(r)(p))(q)$ to q. By (1), this is received by q while in the communication phase of round r, and transferred to its input buffer $inputs_q(p)$. Thus, the corresponding processors of the untimed and time-triggered systems have the same state and input components when they begin the computation phase of round r. The same state transition functions $trans_p$ are then applied by the corresponding processors of the two systems to yield the same values for the corresponding elements of $gu(r+1)$ and $gt(r+1)$, thereby completing the inductive proof.

D. Mechanized Verification

The treatment of synchronous and time-triggered systems in Sections III-A and III-B has been formally specified in the language of the PVS verification system [31], and the verification of Section III-C has been mechanically checked using PVS's theorem prover. The PVS

[6]For the benefit of those not used to reading Curried higher-order function applications, this is decoded as follows: $gu(r)(p)$ is p's state in round r; p's message function msg_p applied to that state gives $msg_p(gu(r)(p))$, which is an array of the messages sent to its outgoing channels; q's component of that array is $msg_p(gu(r)(p))(q)$.

language is a higher-order logic with subtyping, and formalization of the semiformal treatment in Sections III-A and III-B was quite straightforward. The PVS theorem prover includes decision procedures for integer and real linear arithmetic and mechanized checking of the calculations in Section III-C, and the proof of the Theorem, were also quite straightforward. The complete formalization and mechanical verification took less than a day, and no errors were discovered. A description, and the formal specifications and proofs themselves, are available at URL `http://www.csl.sri.com/dcca97.html`.

While it is reassuring to know that the semiformal development of the previous sections withstands mechanical scrutiny, we have argued before (for example, [31, 39]) that mechanized formal verification provides several benefits in addition to the "certification" of proofs. In particular, mechanization supports reliable and inexpensive exploration of alternative designs, assumptions, and constraints. After completing the first version of the work reported here, I wondered whether the requirement that messages be sent at the fixed offset D clocktime units into each round, and that the computation phase begin at the fixed offset P, might not be unduly restrictive. It was the work of a few minutes to generalize the formal specification to allow these offsets to become functions of the round, and to adjust the mechanized proofs. I contend that corresponding revisions to the semiformal development in Sections III-B and III-C would take longer than this, and that it would be difficult to summon the fortitude to scrutinize the revised proofs with the same care as the originals.

IV. Round-Based Algorithms as Functional Programs

The Theorem of Section III-C ensures that synchronous algorithms are correctly implemented by time-triggered implementations that satisfy the various assumptions, constraints, and constructions introduced in the previous section. The next (though logically preceding) step is to ask how one might verify properties of a particular algorithm expressed as an untimed synchronous system.

Although simpler than its time-triggered implementation, the specification of an algorithm as a synchronous system is not especially convenient for formal (and particularly mechanized) verification because it requires reasoning about attributes of imperative programs: explicit state and control. It is generally easier to verify functional, rather than imperative, programs because these represent state and control in an applicative manner that can be expressed directly in conventional logic.

There is a fairly systematic transformation between synchronous systems and functional programs that can ease the verification task by allowing it to be performed on a functional program. I illustrate the idea (which comes from Bevier and Young [23]) using the OM(1) algorithm from Section II. Because that algorithm has already been introduced as a synchronous system, I will illustrate its transformation to a functional program; once the technique becomes familiar, it is easy to perform the transformation in the other direction.

We begin by introducing a function $send(r, v, p, q)$ to represent the sending of a message with value v from processor p to processor q in round r. The value of the function is the message received by q. If p and q are nonfaulty, then this value is v:

$$nonfaulty(p) \wedge nonfaulty(q) \supset send(r, v, p, q) = v,$$

otherwise it depends on the fault modes considered (in the Byzantine case it is left entirely unconstrained, as here).

If T represents the transmitter, v its value, and q an arbitrary receiver, then the communication phase of the first round of OM(1) is represented by

$$send(0, v, T, q).$$

The computation phase of this round simply moves the messages received into the states of the processors concerned, and can be ignored in the functional treatment (though see Footnote 7).

In the communication phase of the second round, each processor q sends the value received in the first round (i.e., $send(0, v, T, q)$) on to the other receivers. If p is one such receiver, then this is described by the functional composition

$$send(1, send(0, v, T, q), q, p). \tag{6}$$

In the computation phase for the second round, processor p gathers all the messages received in the communication phase and subjects them to majority voting.[7] Now (6) represents the value p receives from q, so we need to gather together in some way the values in the messages p receives from all the other receivers q, and use that combination as an argument to the majority vote function. How this "gathering together" is represented will depend on the resources of the specification language and logic concerned: in the treatment using the Boyer-Moore logic, for example, it is represented by a `list` of values [23]. In a higher-order logic such as PVS [31], however, it can be represented by a function, specified as a λ-abstraction:

$$\lambda q : send(1, send(0, v, T, q), q, p)$$

(i.e., a function that, when applied to q, returns the value that p received from q).

Majority voting is represented by a function *maj* that takes two arguments: the "participants" in the vote, and a function over those participants that returns the value associated with each of them. The function *maj* returns the majority value if one exists; otherwise some functionally determined value. (This behavior can either be specified

[7] In the formulation of the algorithm as a synchronous system, p votes on the messages from the *other* receivers, and the message that it received directly from the transmitter, which it has saved in its state. In the functional treatment, q includes itself among the recipients of the message that it sends in the communication phase of the second round, and so the vote is simply over messages received in that round.

axiomatically, or defined constructively using an algorithm such as Boyer and Moore's linear time MJRTY [45].) Thus, p's decision in the computation phase of the second round is represented by

$$maj(rcvrs, \lambda q : send(1, send(0, v, T, q), q, p))$$

where *rcvrs* is the set of all receiver processors. We can use this formula as the definition for a higher-order function $OM1(T, v)$ whose value is a function that gives the decision reached by each receiver p when the (possibly faulty) transmitter T sends the value v:

$$OM1(T, v)(p) \quad (7)$$
$$= \quad maj(rcvrs, \lambda q : send(1, send(0, v, T, q), q, p)).$$

The properties required of this algorithm are the following, whenever the number of receivers is three or more, and at most one processor is faulty.

Requirement 1: Agreement

$$nonfaulty(p) \wedge nonfaulty(q) \supset OM1(T, v)(p) = OM1(T, v)(q),$$

Requirement 2: Validity

$$nonfaulty(T) \wedge nonfaulty(p) \supset OM1(T, v)(p) = v.$$

Definition (7) and the requirements for Agreement and Validity stated above are acceptable as specifications to PVS almost as given (PVS requires us to be a little more explicit about the types and quantification involved). Using a constructive definition for *maj*, PVS can prove Agreement and Validity for a specific number of processors (e.g., 4) completely automatically. For the general case of $n \geq 4$ processors, PVS is able to prove Agreement with only a single user-supplied proof directive, while Validity requires half a dozen (the only one requiring "insight" is a case-split on whether the transmitter is faulty).

Not all synchronous systems can be so easily transformed into a recursive function, nor can their properties always be formally verified so easily. Nonetheless, I believe the approach has promise for many algorithms of practical interest.

V. Conclusion

Many round-based fault-tolerant algorithms can be formulated as synchronous systems. I have shown that synchronous systems can be implemented as time-triggered systems and have proved that, provided care is taken with fault modes, the correctness and fault-tolerance properties of an algorithm expressed as a synchronous system are inherited by its time-triggered implementation. The proof identifies necessary timing constraints and is independent of the particular algorithm concerned; it can be considered a more general and abstract treatment of the analysis performed for a particular system by Di Vito and Butler [46]. The relative simplicity of the proof supports the argument that time-triggered systems allow for straightforward analysis and should be preferred in critical applications for that reason [30].

In recent work, Pfeifer, Schwier, and von Henke of Universität Ulm have formally verified the clock synchronization algorithm used in TTP [41]. Their verification was conducted in PVS and explicitly incorporates the PVS specification, described in Section III-D, that establishes conditions under which synchronous systems can be implemented as time-triggered systems. Thus, in particular, their work provides a mechanically checked formal verification that the TTP clock synchronization algorithm satisfies the four assumptions of Section III-B.

I also showed, by example in Section IV, how a round-based algorithm formulated as a synchronous system can be transformed into a functional "program" in a specification logic, where its properties can be verified more easily, and more mechanically. I have used the same technique to mechanically verify the three-phase commit algorithm (with its termination protocol) [29, Section 7.3.3]. This is a more difficult algorithm than OM(1) and its verification requires proof by induction (in this respect, it is comparable to the r-round algorithm OM(r)), but its representation as a functional program made the mechanized verification quite straightforward and allowed it to be accomplished in a couple of days. Recently, I have verified a group membership algorithm based on [47] (which is related to the group membership algorithm of TTP) using a similar representation. This is a much more challenging exercise and required further methodological development to make it tractable.

I hope this paper has demonstrated that systematic transformations of fault-tolerant algorithms from functional programs to synchronous systems to time-triggered implementations provides a methodology that can significantly ease the specification and assurance of critical fault-tolerant systems. In collaboration with colleagues from Ulm, I am currently applying the methodology to some of the algorithms of TTP [10].

Acknowledgments

Discussions with N. Shankar and advice from Joseph Sifakis were instrumental in the development of this work. Comments by the anonymous referees of both DCCA and TSE improved the presentation considerably.

References

[1] Tushar D. Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost, "On the impossibility of group membership," in *Fifteenth ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, May 1996, Association for Computing Machinery, pp. 322–330.

[2] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[3] Rachid Guerraoui and André Schiper, "Consensus: The big misunderstanding," in *6th IEEE Workshop on Future Trends in Distributed Computing*, Tunis, Tunisia, Oct. 1997, IEEE Computer Society, pp. 183–188.

[4] Hermann Kopetz, *Real-Time Systems: Design Princples for Distributed Embedded Applications*, The Kluwer International

Series in Engineering and Computer Science. Kluwer, Dordrecht, The Netherlands, 1997.

[5] Kenneth Hoyme and Kevin Driscoll, "SAFEbus™," *IEEE Aerospace and Electronic Systems Magazine*, vol. 8, no. 3, pp. 34–39, Mar. 1993.

[6] Aeronautical Radio, Inc, Annapolis, MD, *ARINC Specification 659: Backplane Data Bus*, Dec. 1993, Prepared by the Airlines Electronic Engineering Committee.

[7] William Sweet and Dave Dooling, "Boeing's seventh wonder," *IEEE Spectrum*, vol. 32, no. 10, pp. 20–23, Oct. 1995.

[8] Michael J. Morgan, "Integrated modular avionics for next-generation commercial airplanes," *IEEE Aerospace and Electronic Systems Magazine*, vol. 6, no. 8, pp. 9–12, Aug. 1991.

[9] Akira Hachiga, "The concepts and technologies of dependable and real-time computer systems for Shinkansen train control," in *Responsive Computer Systems*, H. Kopetz and Y. Kakuda, Eds. 1993, vol. 7 of *Dependable Computing and Fault-Tolerant Systems*, pp. 225–252, Springer-Verlag, Vienna, Austria.

[10] Hermann Kopetz and Günter Grünsteidl, "TTP—a protocol for fault-tolerant real-time systems," *IEEE Computer*, vol. 27, no. 1, pp. 14–23, Jan. 1994.

[11] Flaviu Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991.

[12] Flaviu Cristian, Bob Dancey, and Jon Dehn, "Fault-tolerance in air traffic control systems," *ACM Transactions on Computer Systems*, vol. 14, no. 3, pp. 265–286, Aug. 1996.

[13] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.

[14] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev, "Atomic broadcast: From simple message diffusion to Byzantine agreement," in *Fault Tolerant Computing Symposium 15*, Ann Arbor, MI, June 1985, IEEE Computer Society, pp. 200–206, Reprinted in [48, pp. 431–437].

[15] Flaviu Cristian, "Reaching agreement on processor-group membership in synchronous distributed systems," *Distributed Systems*, vol. 4, pp. 175–187, 1991.

[16] Ping Zhou and Jozef Hooman, "Formal specification and compositional verification of an atomic broadcast protocol," *Real-Time Systems*, vol. 9, no. 2, pp. 119–145, 1995.

[17] Yuri Gurevich and Raghu Mani, "Group membership protocol: Specification and verification," in *Specification and Validation Methods*, Egon Börger, Ed., International Schools for Computer Scientists, pp. 295–328. Oxford University Press, Oxford, UK, 1995.

[18] Leslie Lamport and Stephan Merz, "Specifying and verifying fault-tolerant systems," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, H. Langmaack, W.-P. de Roever, and J. Vytopil, Eds., Lübeck, Germany, Sept. 1994, vol. 863 of *Lecture Notes in Computer Science*, pp. 41–76, Springer-Verlag.

[19] John Rushby, "Formal verification of an Oral Messages algorithm for interactive consistency," Tech. Rep. SRI-CSL-92-1, Computer Science Laboratory, SRI International, Menlo Park, CA, July 1992, Also available as NASA Contractor Report 189704, October 1992.

[20] Patrick Lincoln and John Rushby, "Formal verification of an algorithm for interactive consistency under a hybrid fault model," in *Computer-Aided Verification, CAV '93*, Costas Courcoubetis, Ed., Elounda, Greece, June/July 1993, vol. 697 of *Lecture Notes in Computer Science*, pp. 292–304, Springer-Verlag.

[21] Patrick Lincoln and John Rushby, "A formally verified algorithm for interactive consistency under a hybrid fault model," in *Fault Tolerant Computing Symposium 23*, Toulouse, France, June 1993, IEEE Computer Society, pp. 402–411, Reprinted in [48, pp. 438–447].

[22] Patrick Lincoln and John Rushby, "Formal verification of an interactive consistency algorithm for the Draper FTP architecture under a hybrid fault model," in *COMPASS '94 (Proceedings of the Ninth Annual Conference on Computer Assurance)*, Gaithersburg, MD, June 1994, IEEE Washington Section, pp. 107–120.

[23] W. R. Bevier and W. D. Young, "The design and proof of correctness of a fault-tolerant circuit," in *Dependable Computing for Critical Applications—2*, J. F. Meyer and R. D. Schlichting, Eds. Feb. 1991, vol. 6 of *Dependable Computing and Fault-Tolerant Systems*, pp. 243–260, Springer-Verlag, Vienna, Austria.

[24] Chris J. Walter, Patrick Lincoln, and Neeraj Suri, "Formally verified on-line diagnosis," *IEEE Transactions on Software Engineering*, vol. 23, no. 11, pp. 684–721, Nov. 1997.

[25] John Rushby, "A fault-masking and transient-recovery model for digital flight-control systems," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Jan Vytopil, Ed., Kluwer International Series in Engineering and Computer Science, chapter 5, pp. 109–136. Kluwer, Boston, Dordecht, London, 1993, An earlier version is available as [49, pp. 237–257].

[26] Ben L. Di Vito and Ricky W. Butler, "Formal techniques for synchronized fault-tolerant systems," in *Dependable Computing for Critical Applications—3*, C. E. Landwehr, B. Randell, and L. Simoncini, Eds. Sept. 1992, vol. 8 of *Dependable Computing and Fault-Tolerant Systems*, pp. 163–188, Springer-Verlag, Vienna, Austria.

[27] Leslie Lamport, Robert Shostak, and Marshall Pease, "The Byzantine Generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, July 1982.

[28] William D. Young, "Comparing verification systems: Interactive Consistency in ACL2," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 214–223, Apr. 1997.

[29] Nancy A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco, CA, 1996.

[30] Hermann Kopetz, "Should responsive systems be event-triggered or time-triggered?," *IEICE Transactions on Information and Systems*, vol. E76-D, no. 11, pp. 1325–1332, Nov. 1993, Institute of Electronics, Information, and Communications Engineers, Japan.

[31] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS," *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 107–125, Feb. 1995.

[32] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *Journal of the ACM*, vol. 32, no. 1, pp. 52–78, Jan. 1985.

[33] Wilfredo Torres-Pomales, "An optimized implementation of a fault-tolerant clock synchronization circuit," NASA Technical Memorandum 109176, NASA Langley Research Center, Hampton, VA, Feb. 1995.

[34] Paul S. Miner and Steven D. Johnson, "Verification of an optimized fault-tolerant clock synchronization circuit: A case study exploring the boundary between formal reasoning systems," in *Designing Correct Circuits*, Mary Sheeran and Satnam Singh, Eds., Bastad, Sweden, Sept. 1996, Electronic Workshops in Computing (`http://ewic.org.uk/ewic/`).

[35] Frank Schmuck and Flaviu Cristian, "Continuous clock amortization need not affect the precision of a clock synchronization algorithm," in *Ninth ACM Symposium on Principles of Distributed Computing*, Québec City, Québec, Canada, Aug. 1990, Association for Computing Machinery, pp. 133–143.

[36] John Rushby and Friedrich von Henke, "Formal verification of algorithms for critical systems," *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 13–23, Jan. 1993.

[37] Natarajan Shankar, "Mechanical verification of a generalized protocol for Byzantine fault-tolerant clock synchronization," In Vytopil [49], pp. 217–236.

[38] Paul S. Miner, "Verification of fault-tolerant clock synchronization systems," NASA Technical Paper 3349, NASA Langley Research Center, Hampton, VA, Nov. 1993.

[39] John Rushby, "A formally verified algorithm for clock synchronization under a hybrid fault model," in *Thirteenth ACM Symposium on Principles of Distributed Computing*, Los Angeles, CA, Aug. 1994, Association for Computing Machinery, pp. 304–313, Also available as NASA Contractor Report 198289.

[40] D. Schwier and F. von Henke, "Mechanical verification of clock synchronization algorithms," in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lyngby, Denmark, Sept. 1998, vol. 1486 of *Lecture Notes in Computer Science*, pp. 262–271, Springer-Verlag.

[41] Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke, "Formal verification for time-triggered clock synchronization," in *Dependable Computing for Critical Applications—7*, Charles B. Weinstock and John Rushby, Eds., San Jose, CA, Jan. 1999, vol. 12 of *Dependable Computing and Fault Tolerant Systems*, pp. 207–226, IEEE Computer Society.

[42] Herman Kopetz and Wilhelm Ochsenreiter, "Clock synchronization in distributed real-time systems," *IEEE Transactions on Computers*, vol. C-36, no. 8, pp. 933–940, Aug. 1987.

[43] Christopher Temple, "Avoiding the babbling-idiot failure in a time-triggered communication system," in *Fault Tolerant Computing Symposium 28*, Munich, Germany, June 1998, IEEE Computer Society, pp. 218–227.

[44] Carl S. Droste and James E. Walker, *The General Dynamics Case Study on the F16 Fly-by-Wire Flight Control System*, AIAA Professional Study Series. American Institute of Aeronautics and Astronautics, Undated.

[45] Robert S. Boyer and J Strother Moore, "MJRTY—a fast majority vote algorithm," in *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Robert S. Boyer, Ed., vol. 1 of *Automated Reasoning Series*, pp. 105–117. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

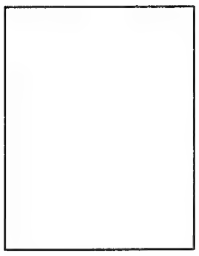
[46] Ricky W. Butler and Ben L. Di Vito, "Formal design and verification of a reliable computing platform for real-time control: Phase 2 results," NASA Technical Memorandum 104196, NASA Langley Research Center, Hampton, VA, Jan. 1992.

[47] Shmuel Katz, Pat Lincoln, and John Rushby, "Low-overhead time-triggered group membership," in *11th International Workshop on Distributed Algorithms (WDAG '97)*, Marios Mavronicolas and Philippas Tsigas, Eds., Saarbrücken Germany, Sept. 1997, vol. 1320 of *Lecture Notes in Computer Science*, pp. 155–169, Springer-Verlag.

[48] IEEE Computer Society, *Fault Tolerant Computing Symposium 25: Highlights from 25 Years*, Pasadena, CA, June 1995.

[49] J. Vytopil, Ed., *Formal Techniques in Real-Time and Fault-Tolerant Systems*, vol. 571 of *Lecture Notes in Computer Science*, Nijmegen, The Netherlands, Jan. 1992. Springer-Verlag.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.



John Rushby received B.Sc. and Ph.D. degrees in computing science from the University of Newcastle upon Tyne in 1971 and 1977, respectively. He joined the Computer Science Laboratory of SRI International in 1983, and served as its director from 1986 to 1990; he currently manages its research program in formal methods and dependable systems. Prior to joining SRI, he held academic positions at the Universities of Manchester and Newcastle upon Tyne in England. His research interests center on the use of formal methods for problems in the design and assurance of dependable systems.

Dr. Rushby is a member of the IEEE, the Association for Computing Machinery, the American Institute of Aeronautics and Astronautics, and the American Mathematical Society. He is an associate editor for these Transactions, and a member of the editorial board for the journal "Formal Aspects of Computing."

Corrected and slightly expanded version of a paper presented at 11th International Workshop on Distributed Algorithms (WDAG '97), Saarbrücken Germany, September 1997. Springer-Verlag Lecture Notes in Computer Science Vol. 1320, pp. 155–169.

Low-overhead Time-Triggered Group Membership*

Shmuel Katz[1,2], Pat Lincoln[1], and John Rushby[1]

[1] Computer Science Laboratory, SRI International, Menlo Park, CA 94025 USA
email: {lincoln, rushby}@csl.sri.com
[2] Computer Science Department, The Technion, Haifa, Israel
email: katz@cs.technion.ac.il

Keywords: time-triggered protocol, group membership, synchronous algorithms, fault tolerance, formal modeling

Abstract. A group membership protocol is presented and proven correct for a synchronous time-triggered model of computation with processors in a ring that broadcast in turn. The protocol, derived from one used for critical control functions in automobiles, accepts a very restrictive fault model to achieve low overhead and requires only one bit of membership information piggybacked on regular broadcasts. Given its strong fault model, the protocol guarantees that a faulty processor will be promptly diagnosed and removed from the agreed group of processors, and will also diagnose itself as faulty. The protocol is correct under a fault-arrival assumption that new faults arrive at least $n+1$ time units apart, when there are n processors. Exploiting this assumption leads to unusual real-time reasoning in the correctness proof.

1 Introduction and Motivation

Group membership has become an important abstraction in providing fault-tolerant services for distributed systems [2]. As in any protocol for group membership, the one presented here allows nonfaulty processors to agree on the membership, and to exclude apparently faulty ones. Because of the strong fault model used, the protocol we consider has the additional desirable properties that the nonfaulty processors agree on the membership at *every* synchronous step, only faulty ones will be removed from the membership, and removal will be prompt. Moreover, a processor with a fault will also diagnose itself promptly.

This protocol for group membership is appropriate for bandwidth-constrained broadcast networks because it requires only one acknowledgment bit to be piggybacked onto existing regularly scheduled broadcasts. The protocol is derived

* This work was supported by Arpa through USAF Electronic Systems Center Contract F19628-96-C-0006, by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044, and by the National Science Foundation under contract CCR-9509931.

from one in a tightly integrated protocol architecture for automobile control [9]. Our contribution is to isolate this group membership protocol (which has not been described explicitly in previous papers), to abstract it from other elements of the integrated protocol, to give a precise formulation of its fault model, and to provide a systematic proof of its correctness. The argument for correctness is interesting and surprisingly intricate because the paucity of information carried in each individual broadcast requires inferences to be made over *sequences* of broadcasts; this, in turn, requires the statement of correctness to be strengthened significantly in order to obtain one that is inductive.

1.1 Background

Algorithms for industrial applications are optimized to deliver maximum utility from minimum resources. These optimizations pose interesting problems in protocol design and analysis that differ from those traditionally considered in the algorithms literature. For example, industrial algorithms for distributed consensus are less concerned with asymptotic reductions in the number of rounds than in maximizing the number of faults that can be tolerated with a small fixed number of rounds (generally two). This leads, for example, to "hybrid" fault models and associated algorithms that permit finer distinctions among faults than purely Byzantine fault models and provide strictly superior fault tolerance [4, 10, 12, 14, 17].

The starting point for the algorithm considered here is the time-triggered protocol (TTP) of Kopetz and Grunsteidl [9]. This protocol is intended for the control of critical functions in automobiles, where communications bandwidth is severely limited, some functions (e.g., ignition timing) require service at very high rates and with great temporal precision, and many functions (e.g., brake-by-wire, steer-by-wire) are safety critical [7]. For these reasons, TTP (and protocols for similar applications, such as ARINC 659 which provides the safety-critical backplane for the top-level avionics functions of the Boeing 777 [1]) are highly integrated, and services such as clock-synchronization, reliable broadcast, group membership, and primary-backup shadowing are combined with the basic data-communication service rather than layered. This allows high-quality services to be provided with very high performance at low overhead (for example, ARINC 659 achieves clock synchronization to within two bit-times at 30 MHz). These protocols also separate fault-tolerance for common cases from those for more severe ones. For example, the group membership protocol of TTP assumes only a single transmit or receive fault (and those faults are themselves narrowly defined) within any two rounds, with more severe fault modes and higher fault arrival rates being handled by a "blackout" operating mode. Empirical data supports these design decisions [9].

Bandwidth is a precious commodity in applications of the family of protocols we study here. Practical considerations such as cost of copper wire, likelihood of failures of interconnects, and lack of skilled maintenance drive designers to focus on simple and cheap hardware interconnect technology such as twisted pair. Extra runs of wire back and forth around a vehicle for redundancy and extra

bandwidth are perceived to be too costly. Wireless communication is viewed as impractical due to the extreme interference expected in the environment. Thus relatively low bandwidth is one of the critical concerns of the designers of these protocols. Even an extra bit per message is considered significant in this domain.

The design constraints on a group membership protocol for an application such as TTP are that it should provide timely and accurate identification and exclusion of faulty processors with minimum overhead. The integrated nature of the protocol means that rather than interpose special "group membership packets" into the communications stream, it should piggyback what it needs for group membership onto the regular data packets. One way to do this is for each processor to append its assessment of the current membership to each packet that it sends. Under suitable assumptions, a protocol can be based on this approach [8], but it is clearly expensive—requiring n bits of membership data appended to each broadcast, for an n processor system (n is 10–20 for these applications). Later descriptions of TTP show only two bits being used for this purpose (actually, they show four, but that appears to be due to the fact that the buses are paired) [9], but the membership protocol is not described. In the following sections, we present a protocol that satisfies these constraints, using only one bit per broadcast, and analyze its properties.

In the following section the model, including its fault assumptions, is first described independently of the group membership problem. Then the assumptions that involve group membership are given, and the kind of reasoning needed for proving correctness is described. The detailed protocol is seen in Section 3 while the proof of correctness is given in Section 4. In the final sections we present a justification for the $n + 1$ limit on fault arrivals, sketch extensions to allow repaired processors to rejoin the group, and briefly describe our use of formal analysis with the Murϕ state exploration system.

The paper shows that a level of abstraction familiar to researchers in distributed programming can be used to isolate and reason about one of a suite of protocols that are combined at the implementation level for efficiency reasons. The separation leads to fault assumptions that seem strong, but are complemented by other assumptions and interleaved protocols.

2 The Model

There are n processors (numbered $0, \ldots, n-1$) arranged in a logical ring and attached to a broadcast bus. Execution is synchronous, with a notional *time* variable increased by one at each step; this, in turn, defines a *slot* in the range $0, \ldots, (n-1)$ as *time* mod n. Nonfaulty processors broadcast whenever it is their slot.

The goal of group membership is to maintain a consistent record of those processors that appear able to communicate reliably and to execute the protocol. A group membership protocol need not tolerate all the types of faults that may afflict the system of which it is a part: other protocols, logically both "above" and "below" group membership, handle some types of faults. In TTP, for example,

replication of the broadcast buses, and strong CRCs (checksums), effectively eliminate message corruption and reduce message loss to a very low level. Clock synchronization ensures that all nonfaulty processors share a common notion of time, and "bus guardians" with independent knowledge of the broadcast schedule prevent faulty processors from speaking out of turn. State-machine replication [16] or pairwise comparison is used to mask or to detect processor faults. These considerations (and empirical measurements) justify considering only two types of faults in the context assumed here.

Send fault: a processor fails to broadcast when its slot is reached.
Receive fault: a processor fails to receive a broadcast.

As noted above, other types of faults can be ignored because they are separately detected by other elements of the total protocol suite and then manifest themselves as either send or receive faults. For example, a transient internal data fault can lead to a processor shutting down and thus exhibiting a send fault when its slot is next reached.

Observe that a send fault can only occur to a processor when it is in the broadcast slot, and a receive fault can only occur to a processor different from the broadcaster. Notice, too, that messages cannot be corrupted, and that a send fault is consistent: *no* processor receives a message from a send-faulty broadcaster. Faults are intermittent: a faulty processor may operate correctly in some steps and manifest its fault in others. A processor is nonfaulty until it manifests a fault, thereafter it is considered faulty; a processor is *actively* faulty at a step if it manifests its fault at that step. That is, a processor is actively send-faulty at a step if it is expected to broadcast but fails to do so; it is actively receive-faulty at a step if it fails to receive the broadcast from a nonfaulty broadcaster.

Two additional assumptions are crucial to the correctness of our protocol, and are justified by the division between a "blackout" operating mode (not considered here) for coping with massive or clustered failures, and the "normal" mode (implemented by the mechanisms described here) that is required to cope only with relatively sparse fault arrivals.

Fault arrival rate: only one nonfaulty processor becomes faulty in any $n + 1$ consecutive slots.
Minimum nonfaulty processors: there are always at least two nonfaulty processors.

The fault model described so far is independent of the problem of group membership. Now we turn to the aspects needed to specify and describe the group membership protocol.

- Each processor has a *local membership set*, that initially contains all processors.
- Processor q is *expected* (to broadcast) by processor p if the current slot is q, and p's local membership set contains q.

A processor will normally broadcast in its slot; it can never broadcast out-of-turn, but it may fail to broadcast in its slot for two reasons:

- It suffers a send fault in that slot,
- It has diagnosed that it suffered an earlier (send or receive) fault and remains silent to inform other processors of that fact.

Using the assumptions and definitions of this model, it is now possible to summarize the requirements specification for the group membership protocol. The required safety property is that the local membership sets of nonfaulty processors are identical in every step, and contain all nonfaulty processors. Additionally, a progress property is needed to exclude trivial solutions: a faulty processor will be removed from the local membership sets of nonfaulty processors no later than the step following its next broadcast slot.[3] Our protocol also ensures that a faulty processor will eventually remove itself from its own membership set (self-diagnosis).

When a processor does broadcast, it appends an "ack" bit to whatever data constitutes the broadcast. This bit indicates whether or not that processor retained the previous expected broadcaster in its membership set. By observing the presence or absence of expected broadcasts, and by comparing the ack bits of received broadcasts to their own observations, processors are able to diagnose their own and other processors' faults and to maintain consistent membership sets.

Non-receipt of an expected broadcast can leave ambiguous the question of whether the transmitter or receiver is faulty. The report (encoded in the ack bit) from the next expected processor is needed to resolve this ambiguity; this report must be reliable, so we will need to show that the next expected processor must be nonfaulty in this case. This does not follow trivially from the fault arrival rate assumption because, for example, the initial non-receipt of a broadcast could be due to that broadcaster falling silent after self-diagnosing a much earlier receive fault. We will need to establish that the diagnosis of faults is sufficiently prompt that it combines with the fault arrival rate assumption to guarantee that the next expected broadcaster cannot be faulty in these circumstances. Thus certain progress properties cannot be separated from the basic agreement properties in the correctness proof.

3 The Protocol

Processors follow a simple fixed procedure: if it is processor p's slot, and p is in its own local membership set, then p attempts to broadcast. (If p is nonfaulty or receive-faulty, it succeeds; if send-faulty, it does not, but is unaware of the fault.) The broadcast includes one bit of state information defined below: the **ack** bit

[3] Technically, the real-time requirement seen here is a safety property and not a progress (or liveness) property in the sense of [11]. However, it does serve to guarantee that needed steps occur and so we refer to it informally as a progress property.

of the broadcaster. Each other processor updates its own local membership set by applying certain rules (described below) to the bit received (or not) from the expected broadcaster. The rules allow each processor to retain or remove either the expected broadcaster or itself from its local membership set, but it will not change its record of membership for any other processor.

Each processor p uses the global variable **time**, a local constant **slot**, and local variables **membership** and **ack**.

- The global variable **time** is an abstraction justified by clock synchronization among local clocks. As noted in the introduction, clock synchronization is assumed to be part of the complete protocol suite along with group membership, and guarantees that all processors agree on the (discrete) value of **time**.
- **slot** is a natural number in the range $0 \ldots n-1$ that records the position of p with respect to other processors in the order of broadcast. This value is fixed and unique for each processor.
- **membership** is the set of processors in p's current view of the group.
- **ack** is a boolean recording whether p received the previous expected broadcast and agreed with the **ack** bit carried in that broadcast. As will be seen shortly, this means that the **ack** bit is true iff p has retained the previous expected broadcaster in its membership, or p was that broadcaster.

We use **ack**(p) to indicate the **ack** bit of processor p, and **slot**(p) to indicate its slot value. Initially, each processor's **membership** contains all other processors, its **ack** is *true*, the global **time** is some initial value (perhaps 0), and each processor is nonfaulty.

The protocol proceeds by discrete time steps; at each step, one processor may broadcast. That broadcaster is the processor b for which **slot**(b) = **time** mod n. The broadcast contains the broadcaster's **ack** bit, plus any data that may be needed for other purposes. The broadcast will be attempted only if b is in its own membership set, and will succeed only if b is not actively send-faulty in that step.

The protocol is described by specifying how each processor p updates its local variables **membership** and **ack** in terms of their previous values, the receipt or non-receipt of an expected broadcast, and the value of the **ack** bit carried in that broadcast.

We first define the auxiliary predicate *arrived*(b, p) as *true* in a step if and only if processor p receives a broadcast from b, and b is the expected broadcaster in that step. This predicate can be considered local to p because that processor can sense the non-receipt of a broadcast.

- For each processor p, if the current broadcaster b is not an element of p's **membership**, none of the local variables are changed.
- If p is the broadcaster b and is in its own **membership** set, it broadcasts **ack**(b) and then updates **ack**(b) to *true*.

- Otherwise, when p is not the broadcaster b, each field of p is updated as follows (notice that $\mathbf{ack}(p)$ is a local variable of p, and that $\mathbf{ack}(b)$ is provided in the broadcast received from b).
  - Updated **membership:** same as previous **membership** except possibly for p and b.
    * p is excluded in two cases:
      (a) (NOT *arrived*(b,p)) AND NOT $\mathbf{ack}(p)$, or
      (b) *arrived*(b,p) AND $\mathbf{ack}(b)$ AND NOT $\mathbf{ack}(p)$.
    * b is excluded in the two cases:
      (c) NOT *arrived*(b,p), or
      (d) $\mathbf{ack}(p)$ AND NOT $\mathbf{ack}(b)$.[4]
  - Updated **ack:** set to *arrived*(b,p) AND ($\mathbf{ack}(b)$ OR NOT $\mathbf{ack}(p)$). Observe that the updated value of $\mathbf{ack}(p)$ is *true* iff p retains b in its local membership (i.e., it is the negation of the disjunction of (c) and (d)). We say that the broadcast by b is *acceptable* to p if the updated value of $\mathbf{ack}(p)$ is *true*.

Thus, p removes itself if (a) two consecutive expected broadcasts are unacceptable, or (b) it considers the previous broadcast unacceptable, but b considers it acceptable. Moreover, p removes b if (c) no broadcast is received or (d) p considers the previous expected broadcast acceptable, while b does not.

The broadcaster always assumes that its broadcast was correctly received even when that was not the case, and thus it sets its **ack** bit to *true*. For other processors, the **ack** bit will be *true* in the following step exactly when the broadcast arrives and either the broadcaster views the previous expected broadcast as acceptable, or the receiver does not.

4 Proof of Correctness

The key safety property of a group membership protocol is agreement: all the membership sets of nonfaulty processors should be identical. Furthermore, all nonfaulty processors should be included in that common membership set. These properties are proved in Theorem 1. The progress property that all faulty processors are promptly diagnosed and removed from the common membership set is proved in Theorem 2, but much of the justification is already present in the invariant required to establish Theorem 1. A corollary is that the common membership set contains at most one faulty processor. In addition, faulty processors are able to diagnose themselves, and do so promptly; this is proved in Theorem 3. These three theorems correspond to the requirements stated in Section 2.

[4] There is a bug in the algorithm as just described: p should exclude itself (not b) when $\mathbf{ack}(p)$ AND NOT $\mathbf{ack}(b)$ and p was the previous broadcaster and sent a *false* **ack** in that broadcast. The bug manifests itself only when where there are exactly three processors in the membership, and its correction affects the proof of Theorem 3. The bug and its correction were pointed out by N. Shankar in an email message on 31 January 1998, and was independently discovered by Sadie Creese and Bill Roscoe of Oxford University using the FDR model checker.

Theorem 1 (Agreement). The local membership sets of all nonfaulty processors are always identical (and are called the *agreed set*) and contain all nonfaulty processors.

This theorem is proved by induction on **time**, but its statement must first be strengthened to yield an assertion that is inductive. In addition to claiming agreement among all nonfaulty processors, and that all nonfaulty processors are included in the agreed membership set, we must claim that all nonfaulty processors have the same values for the **ack** bits at each step, that these bits indeed reflect the intended meaning, and some additional facts about the diagnosis of earlier errors. These are needed to guarantee that in steps in which a fault has been detected, but not yet accurately ascribed, the next expected broadcaster will be nonfaulty and will resolve the uncertainty.

The invariant has the following conjuncts.

(1) All nonfaulty processors have the same local **membership** sets.
(2) All nonfaulty processors are in their own local **membership** sets.
(3) All nonfaulty processors have the same value for **ack**.
(4) For each processor p, $\mathbf{ack}(p)$ is true iff in the most recent previous step in which p expected a broadcast from a processor b, either p was b, or $\mathit{arrived}(b,p) \wedge (\mathbf{ack}(b) \vee \neg\mathbf{ack}(p))$ in that step.
(5) If a processor p became faulty less than n steps ago and q is a nonfaulty processor, either p is the present broadcaster or the present broadcaster is in p's local membership set iff it is in q's.
(6) If a receive fault occurred to processor p less than n steps ago, then either p is not the broadcaster or $\mathbf{ack}(p)$ is *false* while all nonfaulty q have $\mathbf{ack}(q) = \mathit{true}$, or p is not in its local membership set.
(7) If in the previous step b is in the broadcaster slot, p is a nonfaulty processor, and $\mathit{arrived}(b,p)$ does not hold, then b is faulty in the current step.
(8) If the broadcaster b is expected by a nonfaulty processor, then b is either nonfaulty, or became faulty less than n steps ago.

Note that since all nonfaulty processors have identical membership sets and agree on which slot has been reached, they also agree on which processor is the next expected broadcaster. Moreover, by (5), processors that became faulty less than n slots ago agree with the nonfaulty ones on whether the present slot is expected to broadcast. The conjunct (5) is needed to show that newly faulty processors still agree with nonfaulty ones on the next expected broadcaster until they are diagnosed both by others and by themselves.

Conjuncts (7) and (8) are needed to guarantee that no fault has occurred to the processor in an expected slot following one that is newly silent. As mentioned earlier, the prompt diagnosis of receive faults seen in (6) is needed to prove (8). The fault arrival rate assumption thus links the seemingly independent questions of how soon a fault is followed by a (possibly ambiguous) indication that *some* fault has occurred, and how soon after that another fault can occur.

An important feature of the protocol is used in the proof and will be called the *restricted-change* lemma: if a change is made in the local membership set

of p relative to the previous step, it is either in the membership of p itself, or in the membership of the broadcaster in the previous step. This can be seen easily in the description of the protocol. Another useful property that can be seen directly in the description of the protocol is that *arrived*(b, p) will be *true* precisely when $b = p$ or (b is not actively send-faulty, b is in its own membership, p is not actively receive-faulty, and both b and p are in p's membership).

The conjuncts (4) and (7) simply record the intended meanings of **ack** bits and the non-receipt of a broadcast, and follow directly from the assignments to **ack** and the definition of *arrived*, respectively. We show the inductive argument for (5), (6), and (8) separately, and then return to (1), (2), and (3).

Lemma [for conjunct (5)]: If the invariant has been true so far, conjunct (5) will be true of the next step. That is, if in the next step p became faulty less than n steps ago, and q is nonfaulty, then either p is the broadcaster in that step, or the broadcaster is in p's local membership set iff it is in q's.

Proof: Let r denote the broadcaster in the next step. If $p = r$, the lemma holds. Otherwise, $n - 1$ steps ago r was in p's membership iff it was in q's, because both p and q were then nonfaulty and agreed on their membership sets. In all steps since then and up until the next step, r is not the broadcaster and is not p, and thus its membership in p's local membership set is not changed, by the restricted-change lemma. If $q \neq r$, the same reasoning holds for q and r, and we are done. If $q = r$, by the inductive hypothesis, n steps ago the local membership sets of q and p both contained q (when both were nonfaulty), and q still contains itself, since it is nonfaulty in all steps up to the next step, while q is in the membership set of p by reasoning as before. □

Lemma [for conjunct (6)]: If the invariant has been true so far, conjunct (6) will be true of the next step. That is, in the next step, if a receive fault occurred to processor p less than n time ago, then either p is not the broadcaster or **ack**(p) is *false* while all nonfaulty q have **ack**$(q) = true$, or p is not in its local membership set.

Proof: If in the next step p did not become faulty less than n steps earlier, or p is not the broadcaster, the assertion is true. Otherwise, since p became faulty less than n steps earlier, and is now the broadcaster, it was not the broadcaster since it became faulty. Thus until the next step (inclusive) the broadcaster in each step was in p's local membership set if and only if it was in q's, for any nonfaulty q, by (5). If in all previous steps after p became receive-faulty, p and any nonfaulty q did not have the broadcaster at each step in their membership sets, then the **ack** bit of p is *false*. This is true because it was set to *false* in the step it became receive-faulty (by definition of **ack** and a receive fault) and has not been changed since (again, using the definition of **ack** for nonexpected slots). Similarly, for all nonfaulty q in this case, their **ack** bit is *true*: it was set to *true* when p became receive-faulty (since they all did receive a broadcast from

a nonfaulty processor with which they had the same **ack** bit) and has not been changed since. Thus the assertion holds in this case.

If there was a step since p became receive-faulty, but earlier than the next step in which p and any nonfaulty q had the broadcaster r in their local membership sets, then that r must be nonfaulty: by the fault arrival assumption it is not a previously nonfaulty one that is newly send-faulty or receive-faulty within the last n time units (because p has become faulty within the last n), and by the inductive hypothesis, if it had become receive or send-faulty more than n units ago, it would already have been diagnosed in its previous broadcast slot or earlier and thus would not be expected. (Actually, by (6) in the step after its broadcast following its becoming faulty, it would not be in the local membership set of any nonfaulty process.)

So r is nonfaulty and thus will have $\mathbf{ack}(r)$ *true* in its broadcast. If p did not receive that broadcast, then it did not receive two consecutive expected broadcasts and thus removed itself by rule (a). If p did receive the broadcast, it removed itself by rule (b) because it received $\mathbf{ack}(r)$ as *true* while $\mathbf{ack}(p)$ was *false*. Thus in the present step, p is not in its own local membership, as required by the assertion. □

Lemma [for conjunct (8)]: If the invariant has been true so far, conjunct (8) is true in the next step. That is, if the broadcaster in that step p is expected by a nonfaulty processor q, then p is either nonfaulty or became faulty less than n steps ago.

Proof: By contradiction. Consider a situation where broadcaster p is expected by nonfaulty processors q, but p became faulty at least n steps earlier. Then there is an earlier step in which p is the broadcaster, and it became faulty less than n steps earlier. By conjunct (6), if it became receive-faulty, then it will not be in the membership set of any nonfaulty processor in the step following its broadcast (using the conditions for removing a broadcaster), contradicting the fact that it is expected now. If it became send-faulty, it also is not in the membership of any nonfaulty processor in the following step, by rule (c), again contradicting the hypothesis. □

Theorem 1 follows easily from the following claim.

Claim. The conjuncts (1)–(8) are an invariant.

The Proof is by induction.

Basis: All processors are nonfaulty initially and are in all local membership sets, the **ack** bits agree, and there have been no faults.

Inductive step: Conjuncts (5), (6), and (8) have already been proved, while (4) and (7) are simple inductions using the definitions of the terms. Here we show the remaining conjuncts (1)–(3): membership sets of nonfaulty processors agree,

they contain the nonfaulty processors, and the **ack** bits of nonfaulty processors agree.

Assume the invariant is satisfied in all steps up to and including the m'th step (that can be identified with the value of **time** in the state). Consider the $m+1$'st. If the processor at slot (m mod n) is not a member of the agreed set, nothing changes in step $m+1$ except the update of **time**, and the result follows.

Otherwise, the processor at slot (m mod n) is in the agreed set and is expected by nonfaulty processors. If it is nonfaulty, it will broadcast, be received by all nonfaulty processors, and be maintained in their local membership set (the broadcast and local **ack** bits agree by the inductive hypothesis). It also retains itself in its local membership set. All nonfaulty processors will set **ack** to *true* in the next step. No nonfaulty processor will remove itself. This is true because: condition (a) does not hold, since a nonfaulty processor will broadcast and the message is expected and thus is received; condition (b) does not hold because the agreed sets were the same in all previous stages, as were the **ack** bits. Thus all nonfaulty processors still have the same local membership sets and **ack** bits, and include themselves in their local membership sets.

If the processor b at slot (m mod n) is in the agreed set but has a send fault or has detected its own receive fault and removed itself from its local membership set, no nonfaulty processor p will receive b even though it was expected (i.e., $arrived(b,p)$ is *false*), and all will mark it as absent in step $m+1$ by rule (c) and will set **ack** to *false* in that step. No nonfaulty processors will remove themselves in this case: since $arrived(b,p)$ is *false*, condition (b) is irrelevant, and (a) also does not hold, because the neighbors in expected slots around the silent processor must be nonfaulty, by the fault model and the conjuncts (7) and (8). In particular, the broadcast in the most recent expected slot before b was from a nonfaulty processor and thus must have arrived at p and had an **ack** bit that agreed with that of all nonfaulty recipients (by the inductive hypothesis). Therefore the **ack**(p) bit in step m is *true* by conjunct (4). Thus in step $m+1$ the local membership sets and the **ack** bits of those nonfaulty processors remain identical, and no nonfaulty processor removes itself.

If the processor b in the broadcast slot is in the agreed set and in its local membership set (and thus is expected by nonfaulty processors) but is receive-faulty, then by conjunct (8) the receive fault occurred within the last n steps, and by conjunct (6), b will broadcast **ack**(b) as *false*, while nonfaulty processors p have **ack**(p) = *true*. Thus when the new broadcast occurs, all nonfaulty processors will remove the receive-faulty broadcaster by rule (d), and also set **ack** to *false*. In this case too, no nonfaulty processor will remove itself from its local membership set: since $arrived(b,p)$ is *true*, condition (a) is irrelevant, and condition (b) does not hold since the broadcaster had a *false* **ack** bit when it broadcast. □

Most of the justification for prompt diagnosis and removal of faulty processors was provided in the proof of the invariant above. We have:

Theorem 2 (Prompt Removal). A faulty processor is removed from the membership sets of nonfaulty processors in the step following its first broadcast slot while faulty.

Proof: As proved in conjunct (6) of the invariant, if a processor p becomes receive-faulty, then in its next broadcast either $\mathbf{ack}(p)$ is *false*, while $\mathbf{ack}(q)$ is *true* for nonfaulty processors q, or p is not in its local membership set. In the former case, p will be removed from the local membership set of q by rule (d) and in the latter case $arrived(p, q)$ is *false* so that p is removed by rule (c). If p becomes send-faulty, again $arrived(p, q)$ is *false*, so p is removed by rule (c). □

Corollary (One Faulty Member). In any step the agreed group contains at most one faulty processor.

Proof: Immediate from Theorem 2 and the fault arrival rate assumption. □

As part of the proof of the invariant needed for Theorem 1, in conjunct(6), we showed that a processor that is not the next expected broadcaster after becoming receive-faulty will remove itself from its local membership set. Here we show that any faulty processor, including send-faulty ones and those that became receive-faulty just before broadcasting, will remove themselves from their local membership sets.

Theorem 3 (Rapid Self-Diagnosis). A newly faulty processor will remove itself from its local membership set (and thereby diagnose itself) when the slots of at most two nonfaulty processors have been passed.

Proof: If a processor p becomes send-faulty, all nonfaulty processors will set their **ack** bits to *false* in the step following that processor's slot, since the slot is expected and no message is received. Similarly, if p just became receive-faulty in the expected broadcast before its slot, it will broadcast **ack** as *false*, while the nonfaulty processors have $\mathbf{ack} = true$, and thus will set their **ack** bits to *false*. In either case, p will set its **ack** bit to *true* in the step after it broadcasts. Until its own or the previous broadcast, p was nonfaulty, and thus its local membership set agreed with all other nonfaulty processors. By the invariant of Theorem 1, no nonfaulty processor will remove itself due to the new fault, thus the next expected slot of the nonfaulty processors is the same as the next expected slot of the faulty one. By the fault arrival assumption, the next expected slot must be nonfaulty, since it cannot be newly faulty, and by the invariant it cannot be an undiagnosed old fault. Thus the newly faulty processor p will receive $\mathbf{ack} = false$ in the message from the next expected slot, disagree with the broadcaster, and set its own **ack** bit to *false*. Since all nonfaulty processors receive that broadcast and agree with its **ack** bit, p will receive **ack** as *true* in the expected slot after that (using the fact that there are at least two nonfaulty processors within the

group). At that point, the faulty processor p will remove itself from its local membership set, using (b).[5]

If a processor p becomes receive-faulty in its transition to the next step, but p is not the next expected broadcaster, it will remove the broadcaster from its local membership set, but otherwise has the same local membership sets and next expected slot as the nonfaulty ones. It will also set **ack**(p) to *false*. Again by the fault model, the next expected broadcaster must be nonfaulty, will broadcast **ack** as *true*, and the receive-faulty processor p will remove itself using (b). □

5 Discussion and Conclusions

The fault arrival rate we assume in our fault model is at most one new faulty processor in any consecutive $n+1$ slots. This is clearly tight, since if n were used in place of $n+1$, the algorithm fails. Consider a scenario with a receive fault of the processor just before the broadcaster, followed n steps later by a send fault of that same broadcaster. Since the receive-faulty processor will self-diagnose and fall silent in its slot just before the subsequent send fault, all nonfaulty processors will not receive two consecutive expected broadcasts. They will all then incorrectly remove themselves from their local membership sets.

As this analysis shows, clustered faults can cause our algorithm to fail, though it is generally more robust than this worst case analysis suggests: the precise requirements are that the expected broadcaster must be nonfaulty when it follows a receive fault (unless it was that broadcaster that suffered the receive fault in the previous step) or the silence perpetrated by a successfully self-diagnosed receive fault, and the next *two* expected broadcasters must be nonfaulty following a send fault or the broadcast of a processor that suffered a receive fault in the previous step. Stochastic measurements are needed to determine more representative measures of the fault arrival rates and patterns that can be tolerated.

The requirement of two nonfaulty processors is also tight: if there are two processors remaining in the group and one of them becomes faulty, then there is no longer any possibility of distinguishing between a send and a receive fault. In either case, a broadcast is not received by the other processor, each will ultimately remove the other from its local membership set, and neither will ever self-diagnose.

5.1 Robustness of Assumptions

Self-diagnosis requires a very strong assumption on the behavior of faulty processors: namely, that they continue to execute the algorithm correctly for upto n steps after becoming faulty. This is plausible when a send or receive fault is truly due to a communications problem, but is less so when it is the manifestation of a more serious failure. As noted previously, we assume that our group

[5] As noted in footnote 4, this argument is incorrect when there are only three processors. In the corrected algorithm, p excludes itself on the previous round, when it receives **ack** = *false*.

membership protocol is part of a larger suite of protocols that can mask other types of faults, or transform their manifestations into those that can be tolerated by our algorithm. For example, checksums transform corrupted communications into the send and receive faults that our algorithm can tolerate.

Loss of clock synchronization or a "babbling idiot" failure could cause a processor to transmit outside its own slot, thereby possibly preventing nonfaulty processors from communicating on the bus and leading to complete failure of the system. These failure modes are handled using "Bus Guardians" in TTP and self-checking pairs in ARINC 659: in both cases, each processor's access to the bus is mediated by a second component with an independent clock, so that coincident double faults are required to create a catastrophic failure. Computation faults are likewise detected by pairwise comparison, or masked by voting, and higher-level diagnosis and recovery algorithms then isolate or reboot the afflicted processor [15, 18–20]. In these cases, the failure mode is reduced to fail-silence, which manifests itself to our group membership protocol as a combination of long-duration send and receive faults for the processor concerned. Since all nonfaulty processors promptly detect and exclude send-faulty processors without requiring the processor concerned to diagnose itself, the safety properties of group membership are preserved even if the faulty processor is unable to execute the protocol correctly. Self-diagnosis of send and receive faults, which is assured for processors that do correctly execute our protocol, is moot for processors that are in the grip of some larger crisis.

Processor faults that are not masked or detected and reduced to fail-silence by other components of the protocol suite can be catastrophic for group membership if they cause a processor to violate the protocol. For example, if the next expected processor following a send fault incorrectly sends **ack** = *true*, then all other processors will diagnose themselves as receive-faulty and exclude themselves from the group. In TTP, such a collapse causes reversion to the blackout operating mode. More insidious execution faults could allow a processor that suffers a receive fault not to diagnose its failure. Since receive faults (other than of the next expected broadcaster) require the afflicted processor to signal its failure (by remaining silent), such an execution fault would allow the faulty processor to remain a member of the group.

The possibility of such catastrophic or insidious faults is the price paid for the low overhead of our group membership protocol. Experiments with an architecture similar to TTP [6] show that the incidence of such fail-silence violations is sufficiently rare that they present an acceptably low risk to the system (given the other mechanisms present in the total suite of protocols).

5.2 Formal Analysis, and Future Work

We have described and proved correct a protocol for synchronous group membership that, driven by practical considerations, trades a very restrictive fault model in return for very low communications overhead—just one bit per message. Despite the paucity of information carried by each message, the protocol allows rapid and accurate identification and elimination of faulty processors.

The reasoning that supports this claim, however, requires inferences to be made over *sequences* of messages; this, in turn, requires the statement of correctness to be strengthened significantly in order to obtain one that is inductive and requires a surprisingly intricate proof with extensive case-analysis. We found determination of an adequately strong (and true!) invariant, and development of the corresponding proof, to be quite challenging, and turned to formal techniques for assistance. We used the Murϕ state-exploration system [3] to examine instances of the protocol for the purposes of debugging the protocol, its fault model and its assumptions, and also to check candidate invariants. Using Murϕ, we were able to exhaustively check the behaviors of a ring of six processors with up to three faults. This required some tens of minutes (on a Sparc 20) and 100 MB of memory and entailed exploration of almost two million states. We are currently formalizing the general case and subjecting our proof of correctness to mechanical checking using the PVS verification system [13].

For the future, we are interested in systematic techniques for deriving strengthened invariants of the kind needed here, and for generating the proof of correctness. Some of the reasoning resembles that seen in the backward reasoning of precedence properties in temporal logic [11].

The group membership protocol presented here has no provision for readmitting previously-faulty processors that now appear to be working correctly again. Simple extensions, such as allowing a repaired processor to just "speak up" when its slot comes by, are inadequate. (A processor that has a receive fault just as the new member speaks up will not be aware of the fact and its local membership set will diverge from that of the other processors; a second fault can then provoke catastrophic failure of the entire system.) We are aware of solutions that do work, at the cost of strong assumptions on the fault-detection capability of the CRCs appended to each message, and plan to subject these to formal examination. TTP encodes its "critical state" in its CRC calculation, and the ack bit of our abstract protocol is in fact encoded implicitly in the CRC and recovered by recalculation of the CRC for each of the two possible values represented by that bit.

We are also eager to explore more of the highly optimized and integrated algorithms seen in industrial protocols for safety-critical distributed systems, such as TTP and ARINC 659. For example, the restrictive fault model used for our group membership protocol is partly justified by the existence of a blackout operating mode to deal with more severe, or clustered, faults. An interesting challenge for the future is to establish the fault coverage of this combination, and the correctness of the transitions between different operating modes in the presence of faults.

References

Papers by SRI authors are generally available from `http://www.csl.sri.com/fm.html`.

1. *ARINC Specification 659: Backplane Data Bus.* Aeronautical Radio, Inc, Annapolis, MD, December 1993. Prepared by the Airlines Electronic Engineering Committee.
2. Flaviu Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Systems*, 4:175–187, 1991.
3. David L. Dill. The Murϕ verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, NJ, July/August 1996. Springer-Verlag.
4. Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In Ravishankar K. Iyer, Michele Morganti, W. Kent Fuchs, and Virgil Gligor, editors, *Dependable Computing for Critical Applications—5*, volume 10 of *Dependable Computing and Fault Tolerant Systems*, pages 139–157, Champaign, IL, September 1995. IEEE Computer Society.
5. *Fault Tolerant Computing Symposium 25: Highlights from 25 Years*, Pasadena, CA, June 1995. IEEE Computer Society.
6. Johan Karlsson, Peter Folkesson, Jean Arlat, Yves Crouzet, and Güther Leber. Integration and comparison of three physical fault injection techniques. In Brian Randell, Jean-Claude Laprie, Hermann Kopetz, and Bev Littlewood, editors, *Predictably Dependable Computing Systems*, Basic Research Series, pages 309–327. Springer, 1995.
7. H. Kopetz. Automotive electronics—present state and future prospects. In *Fault Tolerant Computing Symposium 25: Special Issue*, pages 66–75, Pasadena, CA, June 1995. IEEE Computer Society.
8. H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avižienis and J. C. Laprie, editors, *Dependable Computing for Critical Applications*, volume 4 of *Dependable Computing and Fault-Tolerant Systems*, pages 411–429, Santa Barbara, CA, August 1989. Springer-Verlag, Vienna, Austria.
9. Hermann Kopetz and Günter Grünsteidl.' TTP—a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.
10. Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society. Reprinted in [5, pp. 438–447].
11. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.
12. Fred J. Meyer and Dhiraj K. Pradhan. Consensus with dual failure modes. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):214–222, April 1991.
13. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
14. John Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Thirteenth ACM Symposium on Principles of Distributed*

Computing, pages 304–313, Los Angeles, CA, August 1994. Association for Computing Machinery.

15. John Rushby. Reconfiguration and transient recovery in state-machine architectures. In *Fault Tolerant Computing Symposium 26*, pages 6–15, Sendai, Japan, June 1996. IEEE Computer Society.
16. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
17. Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *7th Symposium on Reliable Distributed Systems*, pages 93–100, Columbus, OH, October 1988. IEEE Computer Society.
18. C. J. Walter, N. Suri, and M. M. Hugue. Continual on-line diagnosis of hybrid faults. In F. Cristian, G. Le Lann, and T. Lunt, editors, *Dependable Computing for Critical Applications—4*, volume 9 of *Dependable Computing and Fault-Tolerant Systems*, pages 233–249. Springer-Verlag, Vienna, Austria, January 1994.
19. Chris J. Walter. Identifying the cause of detected errors. In *Fault Tolerant Computing Symposium 20*, pages 48–55, Newcastle upon Tyne, UK, June 1990. IEEE Computer Society.
20. Chris J. Walter, Patrick Lincoln, and Neeraj Suri. Formally verified on-line diagnosis. *IEEE Transactions on Software Engineering*, 23(11):684–721, November 1997.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

This article was processed using the LaTeX macro package with LLNCS style

Verifying a Self-Stabilizing Mutual Exclusion Algorithm

To Appear in the Proceedings of PROCOMET'98

Shaz Qadeer
Department of EECS
University of California at Berkeley
Berkeley CA 94720
Phone: (510)642-1490
shaz@eecs.berkeley.edu

Natarajan Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025
Phone: (415)859-5272
shankar@csl.sri.com

Abstract

We present a detailed description of a machine-assisted verification of an algorithm for self-stabilizing mutual exclusion that is due to Dijkstra [Dij74]. This verification was constructed using PVS. We compare the mechanical verification to the informal proof sketch on which it is based. This comparison yields several observations regarding the challenges of formalizing and mechanically verifying distributed algorithms in general.

1 Introduction

A self-stabilizing algorithm is one that ensures that the system behavior eventually stabilizes to a safe subset of states regardless of the initial state. In Dijkstra's original paper [Dij74] introducing distributed self-stabilization, he gave three self-stabilizing mutual exclusion algorithms and left their proofs as exercises for the reader. Subsequently, Dijkstra [Dij86] proffered a proof of self-stabilization of one of his algorithms while conceding that the proofs were in fact nontrivial. In this paper, we take on the challenge of mechanically verifying the self-stability of one of Dijkstra's other algorithms. This algorithm poses an interesting challenging for formalization and mechanical verification for several reasons. The self-stabilization argument involves a nontrivial amount of mathematics. Self-stability is a convergence property whereas typical attempts at the mechanical verifications of distributed algorithms mainly involve safety properties or simple progress reasoning. The use of model checking in the verification of distributed algorithms, even non-finite-state ones, is a recent trend, but there is no obvious way to apply model checking in this context. It is hard to draw substantive conclusions from the verification of one self-stabilization algorithm but we wish to use this example to touch on some of the important themes in formal verification such as:

1. What is the difference in effort between constructing a convincing informal argument and mechanically verifying it?
2. What are some of the specific obstacles to effective formalization?

3. How much of the formal verification involves only conventional mathematics as opposed to temporal reasoning?

4. Can the temporal reasoning be carried out within the framework of a conventional predicate calculus or is there a need for temporal or modal logics?

The heart of this paper is a careful elucidation of the formalization and mechanically verified proof of self-stability for Dijkstra's algorithm. The informal proof of this algorithm is itself quite interesting. A survey of the literature revealed only one attempt at a proof by Varghese [Var92][1] who presents an informal sketch of an argument that is similar to the one used in this paper. As with any distributed algorithm, the verification requires careful attention to details that are easily overlooked in an informal sketch. Our verification was conducted using PVS but our explanation of the verified proof will be presented in a conventional style thus requiring no prior knowledge of PVS [ORS92].[2]

We only consider the first of Dijkstra's three algorithms for self-stabilizing mutual exclusion. In this algorithm, there are N processes for $N > 1$ numbered 0 to $N-1$ arranged in a unidirectional ring where each process can observe its own state and that of its predecessor. Each process i maintains a counter $v(i)$ of values from 0 to $M-1$, where $N \leq M$. Process 0 is a distinguished process that is enabled when $v(0) = v(N-1)$, and when enabled, it can make a transition by incrementing its counter modulo M. A nonzero process i is enabled when $v(i) \neq v(i-1)$, and when enabled, it can update its counter value so that $v(i) = v(i-1)$. The transitions of the individual processes are interleaved. Despite the absence of a central controller, the system can be shown to converge or *self-stabilize* from an arbitrary initial state to a stable behavior where exactly one process is enabled at any state. The one enabled process in a state in such a stable behavior is allowed exclusive access to its critical section.

2 An Informal Proof Sketch

The reader is invited to set aside this paper and contemplate a proof of self-stability for the above algorithm by showing that any computation leading out of an arbitrary initial state eventually stabilizes to a computation where exactly one process is enabled in a state. In our initial verification attempts, we had some difficulty merely constructing a reliable, informal proof sketch. Even with a proof sketch that withstood informal scrutiny, we had a lot of difficulty formalizing the needed concepts in a form that was elegant enough to support feasible mechanical verification. We finally restarted our verification with a proof sketch consisting of the following sequence of claims:

1. *There is always at least one enabled process.* Otherwise, for each i, $i < N-1$, $v(i) = v(i+1)$ yet $v(0) \neq v(N-1)$.

2. *The number of enabled processes never increases.* Each transition due to process i affects the enabledness of at most i and $i+1 \, mod \, N$, and definitely disables i.

3. *The enabledness of each process is eventually toggled.* This is a crucial observation. For each i, let $p(i)$ be the sum over the enabled processes j of the unidirectional distance

[1] See the URL `dworkin.wustl.edu/~varghese/PAPERS/dijk.ps.Z`.

[2] The URL `www.csl.sri.com/pvs/examples/self-stability/` contains the PVS specification and proofs.

from j to i, i.e., $i - j$ for $j \leq i$, and $N - j + i$ for $j > i$. Then, whenever an enabled process j, $j \neq i$, makes a transition, $p(i)$ decreases. If i is initially enabled but never taken, then $p(i)$ eventually becomes 0. By Claim 1, i is the only remaining enabled process, which must therefore be taken leaving i disabled. If i is initially disabled, then when $p(i)$ is 0, it must be enabled by Claim 1.

4. *Process* 0 *is eventually incremented.* This follows trivially from Claim 3.

5. *Process* 0 *eventually takes on any value below* M. Follows by induction from Claim 4.

6. *There is some value* x *below* M *so that for any* i, $i \neq 0$, *we have* $v(i) \neq x$. This is a consequence of the pigeonhole principle which is the key property used to guarantee self-stability.

7. *Eventually* $v(0) = x$ *and for all* i, $i \neq 0$, *we have* $v(i) \neq x$. Let x be chosen as in Claim 6. By Claim 5, eventually $v(0) = x$. Up to (and including) the earliest point where $v(0) = x$, we have $v(i) \neq x$ for any i, $i \neq 0$

8. *Once* $v(0) \neq v(i)$ *for all* i, $i \neq 0$, *then process* 0 *is not enabled again until for all processes* i, $v(i) = v(0)$. This is because given Claim 7, there eventually is a prefix of n processes, for any $n < N$, so that $v(i) = v(0)$ holds for $i \leq n$.

9. *The system eventually reaches a stable state with exactly one enabled process.* By Claim 8, there is eventually a state where only process 0 is enabled, and by Claims 1 and 2, the system will remain stable in a state where there is exactly one enabled process.

Though this informal proof sketch is fairly convincing, the formalization and mechanical verification of Dijkstra's self-stabilizing mutual exclusion protocol using PVS are not straightforward. The formalization and verification are described in Sections 3 and 4.

3 The Formalization of the Problem

Both the algorithm and self-stability property are formalized as predicates on sequences of states. The global *state* in this case consists of an array with indices in the subrange 0 to $N-1$ and with elements in the subrange 0 to $M-1$. A sequence of such states is a function from the type of natural numbers (i.e., non-negative integers) to the *state* type.

A process i is *enabled* in state v when the curried predicate $E(v)(i)$ holds. For process 0, the predicate $E(v)(0)$ holds when $v(N-1) = v(0)$. For a nonzero process i, $E(v)(i)$ holds when $v(i) \neq v(i-1)$. The curried form $E(v)(i)$ is chosen so that we can also employ the predicate $E(v)$ as (the characteristic predicate of) a set of processes.

A sequence of states σ is a function from natural numbers to the *state* type. A sequence of states σ is a *run*, i.e., $run(\sigma)$ holds, when each pair of adjacent states is in U. Two states v and v' are in the next-state relation U (for *update*), i.e., $U(v, v')$, if there exists a process i that is enabled in state v and $UP(i)(v, v')$ holds. The relation $UP(i)(v, v')$ (for *update process*) holds if (1) $i = 0$ and $v'(0) = v(N-1) + 1 \, mod \, M$ and $v'(j) = v(j)$ for $j \neq 0$, or (2) $i \neq 0$ and $v'(i) = v(i-1)$ and $v'(j) = v(j)$ for $j \neq i$.

The variables u, v, v' range over states. The variables x, y, z range over counter values. All increment operations applied to $v(i)$, x, y, z are assumed to be modulo M. The variables

i, j, and k range over process numbers in the subrange 0 to $N-1$. All increment and decrement operations applied to i, j, k are assumed to be modulo N. The various definitions that are used in the proof are summarized in Figure 1.

Term	Definition
$E(v)(i)$	$\begin{cases} v(N-1) = v(0), & \text{if } i = 0 \\ v(i) \neq v(i-1) & \text{otherwise} \end{cases}$
$UP(i)(v, v')$	$\begin{cases} v' = v\{0 \leftarrow v(0)+1\}, & \text{if } i = 0 \\ v' = v\{i \leftarrow v(i-1)\} & \text{otherwise} \end{cases}$
$U(v, v')$	$\exists i: UP(i)(v, v')$
$run(\sigma)$	$\forall n: U(\sigma(n), \sigma(n+1))$
$d(i)(j)$	$\begin{cases} i-j, & \text{if } j \leq i \\ N+i-j, & \text{otherwise} \end{cases}$
$D(v)(i)$	$\sum_{j=0}^{N-1} d(i)(j): j \in E(v)$
$\sigma[n]$	$\lambda\, m:\ \sigma(m+n)$
$S(v)$	$\{x \mid x < M \wedge \exists i: i \neq 0 \wedge v(i) = x\}$
$prefix(v)(j)$	$(\forall i: i \leq j \equiv v(i) = v(0))$

Figure 1: Summary of Definitions

4 The Mechanical Verification

We informally describe the steps in the mechanical verification using PVS. It is worth noting that PVS was used interactively to discover the detailed justifications based on the informal sketch above. The elaboration of the informal sketch is not at all intellectually trivial. First, the formalization, namely the choice of representations and the form of definitions and theorems, has to be carried out with a tremendous amount of care since otherwise the formal justifications can easily become unmanageably large. Second, it can be quite difficult to come up with formal justifications even for informal steps that seem intuitively obvious. It is this latter aspect of the informal to formal transition that we wish to highlight in presenting the details of the mechanical verification below.

Initial proof attempts such as the ones described below tend to be verbose as they represent the form in which the proofs were discovered. It is customary to devote some effort to polishing PVS proofs so as to enhance their conciseness while exploiting the available automation. We indicate the number of interactions needed to construct the mechanically checked proof of individual formulas in order to convey a rough idea of the effort needed. This number does not have any precise significance since some interactions consist of commands with many inputs that invoke complex proof strategies, while others consist of simple atomic

inference steps. Note that the verification relies on pre-existing PVS libraries for the *mod* operation and for finite cardinalities.[3]

Lemma 4.1 corresponds to Claim 1.

Lemma 4.1 *There is always at least one enabled process. Formally,* $\forall v : \exists i : E(v)(i)$.

Proof. Suppose that $\forall j: v(j) = v(0)$, then in particular, $v(N-1) = v(0)$, but this would imply that process 0 is enabled. Otherwise, when $\forall j: v(j) = v(0)$ does not hold, there is a least j, $j \neq 0$ such that $v(j) \neq v(0)$ and since $v(j) \neq v(j-1)$, process j is enabled. The second part of this proof is actually formally proved by induction on j.

The mechanization of this entire proof consists of ten interactions. The first two steps are universal quantifier elimination and case splitting. The first case involves five interactions for existential quantifier instantiation and simplification. The second case involves three interactions for induction/simplification and quantifier instantiation followed by simplification. ∎

Lemma 4.2 is a step in the proof of Lemma 4.3 which corresponds to Claim 2.

Lemma 4.2 *An update to an enabled process i removes i and could otherwise at most insert or remove $i+1$ from the set of enabled processes. Formally,*

$$UP(i)(v,v') \supset E(v') = \begin{cases} E(v) - \{i\} \bigcup \{i+1\}, & \text{if } E(v')(i+1) \\ E(v) - \{i, i+1\}, & \text{otherwise} \end{cases}$$

Proof. There are two parts to this proof. First, we establish that $i \notin E(v')$. This is trivial when $i \neq 0$ since $v'(i) = v'(i-1)$. When $i = 0$, a previously proved lemma yields $v(0)+1 \neq v(0) \, mod \, M$ which is used to show that $v'(0) \neq v'(N-1)$. The second part of the proof essentially demonstrates that only the enabledness of i and $i+1$ are affected when i is updated. The extensionality rule for set equality is applied to the conclusion which then follows from the lemmas $x < N \supset x \, mod \, N = x$ and $N \, mod \, N = 0$ by definition expansion and simple set-theoretic reasoning.

The mechanization has three preliminary interactions for quantifier elimination, definition expansion (for *UP*) and propositional simplification. The next step introduces the case analysis for splitting the proof into two parts. The first part is propositionally simplified into two cases. The first case is proved by introducing a lemma instance followed by simplification and definition expansion. The second case is proved directly by expansion/simplification. The second part of the proof has three interactions for using extensionality, introducing a lemma instance, and brute-force expansion/simplification. The last brute-force step is not entirely pleasant since a lot of essential detail has been buried in the automation. A more careful argument would be to show that when $UP(i)(v,v')$, then for $j \notin \{i, i+1\}$, $j \in E(v')$ iff $j \in E(v)$. ∎

Lemma 4.3 *A transition never increases the cardinality of the set of enabled processes. Formally,* $U(v,v') \supset |E(v')| \leq |E(v)|$.

Proof. This lemma is of course a simple consequence of Lemma 4.2. There is, however, one minor complication. The cardinality operation applies only to finite sets and the sets $E(v)$ and $E(v')$ have to be shown to be finite. The typechecker generates proof obligation

[3]The URL `www.csl.sri.com/pvs.html` contains links relevant to PVS including pointers to the groups using PVS, several example verifications, and an extensive bibliography of papers by PVS users.

to this effect. This proof obligation is discharged by exhibiting an injection from these sets to some bounded initial segment of the natural numbers. Since the elements of $E(v)$ are already chosen from the subrange of numbers below N, we can choose N as the bound and the identity function on these sets as the required injection. The proof of the lemma follows from Lemma 4.2 using the above proof obligations (which have to be proved separately) as lemmas. This proof also invokes simple cardinality lemmas regarding the operations of adding and removing elements from a finite set: $|E(v) - \{i\}| = |E(v) - 1|$ when $i \in E(v)$, and hence, $|(E(v) - \{i\}) \bigcup \{i+1\}| \leq |E(v)|$.

The mechanization uses three interactions to bring in Lemma 4.2 and the finiteness proof obligations. A single GRIND step, the catch-all brute-force strategy[4] completes most of the proof while leaving one trivial subgoal. This is however quite misleading because the arguments to GRIND are quite specific about which lemmas and definitions are to be used, and which ones excluded. The main observation is that though such reasoning about cardinalities is entirely trivial in informal terms, there are a number of bookkeeping details that are needed in the formal proof to ensure that all the sets involved are indeed finite. ∎

We now formally define the measure that is used in Claim 3 to demonstrate that each process is always eventually enabled and eventually disabled. The formalization deviates slightly from the informal script. We can avoid the case analysis in Claim 3 but use the very same measure to show the equivalent fact that each process is always eventually updated. The measure is defined in terms of a summation operation $\sum_{i=0}^{n} f(i) : i \in P$ which takes a function f, a set P, and an upper bound n and adds up the $f(i)$ for $0 \leq i \leq n$ such that $i \in P$ holds. It is easy to prove the following lemmas by induction on n followed by a handful of simplification steps.

Lemma 4.4

$$\sum_{i=0}^{n} f(i) : i \in (P - \{j\}) = \begin{cases} \sum_{i=0}^{n} f(i) : i \in P - f(j), & \text{if } j \leq n \wedge j \in P \\ \sum_{i=0}^{n} f(i) : i \in P, & \text{otherwise} \end{cases}$$

Lemma 4.5

$$\sum_{i=0}^{n} f(i) : i \in (P \bigcup \{j\}) = \begin{cases} \sum_{i=0}^{n} f(i) : i \in P, & \text{if } j > n \vee j \in P \\ \sum_{i=0}^{n} f(i) : i \in P + f(j), & \text{otherwise} \end{cases}$$

Let $d(i)(j)$ be the unidirectional distance from i to j defined as $i - j$ when $j \leq i$ and $N + i - j$, otherwise.

Then for any process i, the measure for i in state v is given by $D(v)(i)$ which is defined as $\sum_{j=0}^{N-1} d(i)(j) : j \in E(v)$. In words, this measure is the sum of the unidirectional distance of the enabled processes from i in state v. This simple definition of the measure function was constructed after several failed attempts at mechanization. In our earlier attempts, we tried to define the measure by summation over the processes k steps away from j and this made all the inductions very complicated.

Lemma 4.6 *The measure $D(v)(j)$ for j decreases with any update to an enabled non-j process. Formally,*

$$i \neq j \wedge UP(i)(v, v') \supset D(v')(j) < D(v)(j).$$

[4]The GRIND strategy performs quantifier elimination and instantiation, propositional simplification, rewriting using lemmas as rewrite rules, definition expansion, explicit case analysis according to the case structure in the goal, and does many of these steps repeatedly until no further simplification is possible.

Proof. First, note that $i \in E(v)$. By Lemma 4.2, the goal simplifies to considering $E(v')$ to either be $E(v) - \{i\} \bigcup \{i+1\}$ or $E(v) - \{i\}$. In the first case, we have a further case analysis according to whether $j = i+1$. If $j = i+1$, then since $i \in E(v)$, we have by Lemmas 4.2 and 4.4 that $D(v')(j)$ is $D(v)(j) - d(j)(i) + 0$. Since $i \neq j$, we have $d(j)(i) > 0$ and $D(v')(j) < D(v)(j)$. If $j \neq i+1$, then by Lemmas 4.2, 4.4, and 4.5, $D(v')(j)$ is at most $D(v)(j) - d(j)(i) + d(j)(i+1)$. We can prove that when $i \neq j$, $d(j)(i) > d(j)(i+1)$ and hence $D(v')(j) < D(v)(j)$. In the second case when $E(v')$ is $E(v) - \{i\}$ and since $i \in E(v)$ and $D(v')(j) = D(v)(j) - d(j)(i)$, we again have by $d(j)(i) > 0$ that $D(v')(j) < D(v)(j)$.

The mechanical verification carries out a similar case analysis. Of the four cases, three are trivially discharged by the GRIND strategy and the remaining case has three further steps using a lemma about the *mod* operation. ∎

Lemma 4.7 *No non-j process is enabled when the measure for j is 0 in state v. Formally,* $D(v)(j) = 0 \supset (\forall i: i \neq j \supset i \notin E(v))$.

Proof. This is proved by first expanding the definition of D in terms of summation and generalizing the theorem to bound the summation by some n that is below N (rather than $N-1$) so that the theorem can now be proved by induction. The induction proof is then straightforward since if $i \neq j$, then $d(j)(i) > 0$ so if the summation $\sum_{k=0}^{n+1} d(j)(k): k \in E(v)$ equals 0, then if $i = n+1$ it must be because $n+1 \notin E(v)$. If $i \leq n$, then since $\sum_{k=0}^{n} d(j)(k): k \in E(v)$ equals 0, the induction hypothesis can be used to demonstrate that $i = j$ or $i \notin E(v)$.

The mechanical proof requires ten interactions that follow the above outline. The default brute-force induction strategy fails primarily because the definition of summation defeats a heuristic that bounds the depth to which recursive definitions are expanded, and also the instantiation heuristic is unable to find the correct instantiation for the quantified variables. ∎

Up to this point, we have merely proved basic mathematical facts or theorems about a single transition step in a conventional mathematical style. We have not introduced any temporal properties such as invariants or eventualities. We now show the first temporal property (Claim 3) that a process i is eventually updated in any run of the system. A run of the system is a sequence σ such that $\forall n: U(\sigma(i), \sigma(i+1))$ holds. Note that any suffix of a run is also a run. The statement of the theorem is that if σ is a run, then $\exists n: UP(i)(\sigma(n), \sigma(n+1))$. We did this proof in two different ways. In the first attempt, we proved it directly by measure induction on the measure function. In the second case, we established an eventuality rule that was used to prove the goal. Somewhat unexpectedly, the latter proof attempt turned out to be less straightforward than the first one. The main reason is that the overhead of using the eventuality rule outweighed the savings. We explain the verification using the eventuality rule below.

Let the suffix of σ which is the sequence $\sigma(n), \sigma(n+1), \ldots$ be denoted by $\sigma[n]$. The eventuality rule establishes $\exists n: R(\sigma[n])$ for a given trace predicate R by requiring a measure function f on a trace with a well-founded ordering on the range type (such as the usual $<$ relation on natural numbers) such that for any run θ, either $R(\theta)$ or $f(\theta[1]) < f(\theta)$.

Theorem 4.8 *If there is a function f from sequences to natural numbers such that*

$$\forall \theta: run(\theta) \supset R(\theta) \vee f(\theta[1]) < f(\theta)$$

then

$$\forall \sigma: run(\sigma) \supset \exists n: R(\sigma[n]).$$

Proof. The conclusion is proved by measure induction on $f(\sigma)$. By the premise, $R(\sigma)$ or $f(\sigma[1]) < f(\sigma)$. When $R(\sigma)$ holds, we already have the conclusion $R(\sigma[0])$ since $\sigma[0] = \sigma$. When $f(\sigma[1]) < f(\sigma)$ holds, we can apply the induction hypothesis on $\sigma[1]$ to get $R(\sigma[1][n])$ which is the same as $R(\sigma[n+1])$ and hence the conclusion.

The eventuality rule is easily proved in about thirteen steps mainly for invoking measure induction and providing quantifier instantiations, expanding definitions, and applying extensionality to demonstrate equality between sequences. ∎

Lemma 4.9 *Each process i is eventually updated in any run. Formally, $run(\sigma) \supset \exists n\colon UP(i)(\sigma(n), \sigma(n+1))$.*

Proof. With the eventuality rule, we can avoid measure induction in the proof that in any run σ, process i is eventually updated, i.e., $\exists n\colon UP(i)(\sigma(n), \sigma(n+1))$. This removes one major intellectual obstacle from the proof. We still need to identify the measure in terms of σ. In this case, the measure is $D(\sigma(0))(i)$. Four interactions are needed to introduce and properly instantiate the eventuality rule. Four more interactions are needed to show that the conclusion of the eventuality rule $run(\sigma) \supset \exists n\colon R(\sigma[n])$ can be instantiated to yield the conclusion $run(\sigma) \supset \exists n\colon UP(i)(\sigma(n), \sigma(n+1))$. The corresponding premise of the eventuality rule

$$UP(i)(\sigma(0), \sigma(1)) \vee D(\sigma(1))(i) < D(\sigma(0))(i)$$

can be proved by first expanding *run* to get $\exists j\colon UP(j)(\sigma(0), \sigma(1))$. If $i = j$ then we have $UP(i)(\sigma(0), \sigma(1))$. Otherwise, we have to invoke Lemma 4.6 to get $D(\sigma(1))(i) < D(\sigma(0))(i)$. The mechanical proof of this use of the eventuality rule takes about twelve interactions. Overall, the proof of this lemma takes twenty interactions, whereas the direct proof of this theorem by measure induction and without using the eventuality rule took 21 interactions. ∎

Lemma 4.10 corresponds to Claim 4.

Lemma 4.10 *Process 0 is eventually incremented in any run. Formally,*

$$run(\sigma) \supset \exists n\colon \sigma(n)(0) = \sigma(0)(0) + 1.$$

Proof. This claim is a consequence of Lemma 4.9 but not obviously so. We know from Lemma 4.9 that process 0 will eventually be updated but only the first such update will lead to the desired increment. We also need to ensure that none of the intervening transitions affects the state of process 0. If σ is a run, let m be given by Lemma 4.9 so that $UP(0)(\sigma(m), \sigma(m+1))$ holds. We instead prove that

$$\forall m, \sigma\colon run(\sigma) \wedge UP(0)(\sigma(m), \sigma(m+1)) \supset \exists n\colon \sigma(n)(0) = \sigma(0)(0) + 1$$

by induction on m. If m is 0 then we can let n be 1 since by the definition UP, $\sigma(1)(0) = \sigma(0)(0) + 1$. In the induction step when $m > 0$, we know by the definition of *run* that $UP(i)(\sigma(0), \sigma(1))$ holds for some i. If i is 0, then once again, we can let n be 1 since by the definition of UP, $\sigma(1)(0) = \sigma(0)(0) + 1$. If $i \neq 0$ then $\sigma(1)(0) = \sigma(0)(0)$. Since in $\sigma[1]$, the occurence of $UP(0)(\sigma(m), \sigma(m+1))$ is reachable in $m-1$ steps, we can instantiate the induction hypothesis by $\sigma[1]$. Since we know that $run(\sigma[1])$ and $UP(0)(\sigma[1](m-1), \sigma[1](m))$ hold, we can conclude by the induction hypothesis that there is an n' such that $\sigma[1](n')(0) = \sigma[1](0)(0) + 1$ and since $\sigma(1)(0) = \sigma(0)(0)$ we have $\sigma(n'+1)(0) = \sigma(0)(0) + 1$ yielding $n'+1$ as the desired witness for n.

The mechanical proof essentially follows each step of the above outline. The proof could not be carried out more automatically since some of the quantifier instantiations were not found by the simple syntactic matching techniques used by PVS to instantiate quantifiers. This is the first example of a theorem that is fairly obvious but whose proof requires a fair amount of formal machinery for extracting the induction scheme that underlies the intuitive reasoning and for carefully instantiating quantifiers. ∎

Lemma 4.10 can be used to prove Lemma 4.11 which corresponds to Claim 5.

Lemma 4.11 *Process* 0 *can eventually take on any counter value. The formal statement is*

$$\forall x: run(\sigma) \supset \exists n: \sigma(n)(0) = x.$$

Proof. The proof is by measure induction on the number of increments needed to reach x from $\sigma(0)(0)$. If none are needed, then we can clearly choose n to be 0. Otherwise, by Lemma 4.10 we have an n' such that $\sigma(n')(0) = \sigma(0)(0) + 1$. We can now apply the induction hypothesis to $\sigma[n']$ since we one fewer increment is needed to reach x. By the induction hypothesis, we have an n'' such that $\sigma[n'](n'')(0) = x$. The witness n for the main conclusion then can be chosen to be $n' + n''$.

The primary points where the mechanical proof has to be carried out delicately are in some of the definition expansions, the quantifier instantiations, and in some steps involving properties of *mod*. Although the proof involves 27 interactions, many of these involve simple but important details such as type correctness proof obligations. The outline of the proof matches the above description and the details are quite straightforward. ∎

The pigeonhole principle (Claim 6) is a classic example of a theorem that is substantially more obvious than its proof. The principle can be stated in many forms but the one that is useful to us is shown below. Informally, the principle states that if each of $n + 1$ pigeons is assigned one of n pigeonholes, then some hole must contain at least 2 pigeons.

Lemma 4.12 (Pigeonhole principle) *There is no injection from the segment of numbers below* $n + 1$ *to the segment of numbers below* n.

Proof. The proof is by contradiction using a straightforward induction on n. If $n = 0$ then the range type of the required injection is empty so there is no function to such a range type. Otherwise, if $n \neq 0$ and there is an injection f from the segment below $n+1$ to the segment below n, then we can construct an injection from the segment below n to the segment below $n - 1$ as follows. Define a function g over the domain type of the segment below n so that $g(k)$ is $f(k)$ for $f(k) < f(n)$ and $f(k) - 1$ otherwise. The range type of g is the segment below $n - 1$. The function g can be shown to also be an injection by the the case analysis used in its definition. This contradicts the induction hypothesis that there is no injection from the segment below n to the segment below $n - 1$.

The primary creative input in the mechanical proof is the definition of g. Otherwise, the proof is a straightforward induction followed by an easy case analysis. This proof required 16 interactions.[5] ∎

The following is yet another example of a seemingly obvious theorem that does not have a simple formal argument. Let $S(v)$ denote the set $\{x | x < M \wedge \exists i: i \neq 0 \wedge v(i) = x\}$.

[5]The PVS proof should be compared with a similar proof carried out by Boyer and Moore [BM84] using the Boyer-Moore theorem prover. In the latter proof, the injection is represented as a list with some complicated conditions that constrain it to behave as an injection.

Lemma 4.13 *The nonzero processes do not contain all the possible counter values. The formal statement of the theorem is* $\exists x: x < M \wedge x \notin S(v)$.

Proof. The intuitive reason is that there are at most $N-1$ nonzero processes and hence at most $N-1$ distinct counter values. If there are M possible counter values where $N \leq M$, then there must be some x below M such that $x \notin S(v)$.

A detailed description of the mechanized proof is as follows. First, there exists an injection f from $S(v)$ to the segment below $N-1$. Such a function f can be defined as the inverse of the mapping from i to $v(i+1)$ since $S(v)$ is the image set of such a mapping. Note that the inverse function f^{-1} is defined on the image of f so that $f^{-1}(j) = i$ for some i such that $f(i) = j$. An inverse of any mapping is injective on the image of the mapping.

If the conclusion of the theorem is false, then $S(v)$ coincides with the segment below M and the identity function I is an injection from the segment below M to $S(v)$. Since the composition of injections is an injection, we have an injection $f \circ I$ from the segment below M to the segment below $N-1$ and hence the segment below $M-1$. This contradicts the pigeonhole principle (Lemma 4.12).

The mechanical proof carries out the above construction but a large number of type correctness proof obligations are generated due to the sophisticated use of predicate subtyping. The proof has 51 interactions out of which all but 15 interactions deal with simple type correctness proof obligations. ∎

At this point we have established that there is a counter value that does not occur in the nonzero processes and we also know that through successive increments, process 0 can take on any counter value. The next step is to show (Claim 7) that process 0 can acquire a counter value that does not occur in any of the nonzero processes, and furthermore once this value is acquired, it is propagated along a growing prefix of processes until all processes have the same counter value (Claim 8). At this point, process 0 is the only enabled process and the system has entered a stable state.

Let $\mathit{prefix}(v)(j)$ be defined as $(\forall i: i \leq j \equiv v(i) = v(0))$. The first step is to show that process 0 can acquire a counter value that is different from the counter value of any other process. We know by Lemma 4.13 that there is a value x below M that does not occur as the counter value of any nonzero process. We know by Lemma 4.11 that process 0 does eventually take on value x in the nth state of the computation, for some n, but the lemma does not imply that no other process has value x at this point. We need to prove the statement that 0 acquires a counter value that is distinct from those of other processes by induction on the number of steps n given by Lemma 4.11.

Lemma 4.14 *In any run* σ*, eventually process* 0 *acquires a counter value that is distinct from those of the nonzero processes. Formally,*

$$\forall n, \sigma: \mathit{run}(\sigma) \wedge \sigma(n)(0) = x \wedge x \notin S(\sigma(0)) \supset (\exists n': (\forall i: i = 0 \equiv \sigma(n')(i) = x)).$$

Proof. Note that the formal statement has an extra antecedent asserting that $\sigma(n)(0) = x$ which we know can be discharged by Lemma 4.11, but is required here since the proof is by induction on n. In the base case when $n = 0$, the proof follows easily by letting n' be 0 since $\sigma(0)(0) = x$ but $x \notin S(\sigma(0))$. In the induction step, when $n > 0$ we instantiate σ in the induction hypothesis with $\sigma[1]$. This yields an induction hypothesis

$$\begin{aligned} &\mathit{run}(\sigma[1]) \wedge \sigma[1](n-1)(0) = x \wedge x \notin S(\sigma[1](0)) \\ \supset\ &(\exists n'': (\forall i: i = 0 \equiv \sigma[1](n'')(i) = x)). \end{aligned}$$

If $x \in S(\sigma[1](0))$, then it must be the case $\sigma(0)(0) = x$ and the transition from $\sigma(0)$ to $\sigma(1)$ is on process 1. In this case, we let n' be 0 since $\sigma(0)(0) = x$ but $x \notin S(\sigma(0))$. Otherwise, when $x \notin S(\sigma[1])$, we obtain an n'' from the induction hypothesis so that $(\forall i: i = 0 \equiv \sigma[1](n'')(i) = x)$. We let n' be $n'' + 1$ to prove the required conclusion.

The corresponding PVS proof is constructed with 25 interactions. ∎

The next claim is needed in the induction step for showing that if we have a homogeneous prefix from 0 to j where $j < N - 1$, then it can be extended to a homogeneous prefix from 0 to $j + 1$. Note that the claim in the previous sentence is incorrect unless the definition of *prefix* ensures that the prefix value does not occur outside the prefix.

Lemma 4.15 *If process $j + 1$ is eventually updated in a run where there initially is a homogeneous prefix from 0 to j, a homogeneous prefix from 0 to $j+1$ is eventually obtained. Formally,*

$$\begin{aligned} &\forall n, \sigma: run(\sigma) \wedge UP(j+1)(\sigma(n), \sigma(n+1)) \wedge prefix(\sigma(0))(j) \\ &\quad \supset \quad (\exists n': prefix(\sigma(n'))(j+1)). \end{aligned}$$

Proof. This claim is proved by induction on n. When $n = 0$, $UP(j+1)(\sigma(0), \sigma(1))$ holds. If we take n' to be 1, then the conclusion follows by noting that $\sigma(1)(j+1) = \sigma(0)(j) = \sigma(0)(0)$ and for $i > j + 1$, $\sigma(1)(i) = \sigma(0)(i) \neq \sigma(0)(0)$. The quantifier instantiation in the proof of the base case is quite tricky and the mechanical proof requires six interactions.

In the induction step when $n > 0$, we have to show that the conclusion follows from the induction hypothesis

$$\begin{aligned} &\forall \sigma': run(\sigma') \wedge UP(j+1)(\sigma'(n-1), \sigma'(n)) \wedge prefix(\sigma'(0))(j) \\ &\quad \supset \quad (\exists n'': prefix(\sigma'(n'')(j+1))). \end{aligned}$$

We have by $run(\sigma)$ that there is some k such that $UP(k)(\sigma(0), \sigma(1))$ holds. This k cannot be in the subrange from 0 to j since these processes are not enabled in $\sigma(0)$. If $k = j+1$, then we can repeat the reasoning carried out in the base case. If $k > j+1$, then we instantiate the induction hypothesis with $\sigma[1]$ for σ'. Clearly $run(\sigma[1])$ and $UP(j+1)(\sigma[1](n-1), \sigma[1](n))$ follow from the corresponding formulas in the induction conclusion. The antecedent of the induction hypothesis, $prefix(\sigma[1](0))(j)$, also follows from the corresponding formula since $\forall i: i < j \supset \sigma(1)(i) = \sigma(0)$. The induction hypothesis therefore yields an n'' such that $prefix(\sigma[1](n''))(j+1)$ holds. We can therefore prove the induction conclusion by instantiating n' with $n'' + 1$.

This entire proof is fairly sizable and requires 33 interactions. ∎

Lemma 4.16 *In any run, it is always possible to reach a state where the processes from 0 to j have the same counter value and this counter value does not occur in the processes numbered from $i + 1$ to $N - 1$. The formal statement is*

$$\forall j, \sigma: run(\sigma) \supset \exists n: prefix(\sigma(n))(j).$$

Proof. The proof is by induction on j. In the base case, when $j = 0$, we know by Lemma 4.13 that there is at least one x such that $x \notin S(\sigma(0))$. We know by Lemma 4.11 that there is an n'' where $\sigma(n'')(0) = x$. We can use Lemma 4.14 with n'' for n to get an n' such that $prefix(\sigma(n'))(0)$ holds.

Next we turn to the induction step when $j > 0$. The formula to be proved here is

$$run(\sigma) \wedge prefix(j-1)(\sigma(n)) \supset (\exists m{:}\, prefix(\sigma(m))(j)).$$

By instantiating σ in Lemma 4.9 with $\sigma[n]$ we get an n'' at which process j is updated and $UP(j)(\sigma[n](n''), \sigma[n](n''+1))$ holds. We can therefore apply Lemma 4.15 with $\sigma[n]$ for σ, n'' for n, and $j-1$ for j, to obtain an n' where $prefix(\sigma[n](n'))(j)$ holds. Thus, $n+n'$ serves a witness for m in our desired conclusion. ∎

At this point, we have all the ingredients needed to prove the self-stabilization property (Claim 9). Again, one might think that this is an obvious consequence of the claims presented thus far but the details are by no means straightforward.

Theorem 4.17 *In any run, there is a point beyond which the cardinality of the set of enabled processes is exactly one. The theorem is formally stated as*

$$\forall \sigma{:}\, run(\sigma) \supset (\exists m{:}\, \forall n{:}\, |E(\sigma[m](n))| = 1).$$

Proof. The first step in the proof is to invoke Theorem 4.16 with $N-1$ for j to obtain a state $\sigma(m)$ where the counter values of all the processes are identical. This leaves us with the task of proving that if $prefix(\sigma[m](0))(N-1)$ then $\forall n{:}\, |E(\sigma[m](n))| = 1$. This is proved by induction on n. In the base case, we first show that $E(\sigma[m](0)) = \{0\}$. This follows by applying extensionality and using the homogeneous prefix property twice: once on i to show that for any i, if $i \in E(\sigma[m](0))$ then $i = 0$, and again on 0 to show that $0 \in E(\sigma[m](0))$. The second step is a trivial one of showing that the cardinality of the set $\{0\}$ is 1, but it requires the use of some lemmas concerning finite cardinalities. The mechanical verification of the base case takes ten interactions.

In the induction step, we need to show that the cardinality of enabled process remains 1 under a transition from $\sigma[m](n)$ to $\sigma[m](n+1)$. We know by Lemma 4.3 that $|E(\sigma[m](n+1))| \leq |E(\sigma[m](n))|$ which means $|E(\sigma[m](n+1))|$ is either 0 or 1. It cannot be 0, since this contradicts Lemma 4.1 which asserts that there is always at least one enabled process. Hence the conclusion. The induction step takes up about eleven interactions, and about eight more interactions are needed to discharge the type-correctness proof obligations that come up, particularly those that require the demonstration that certain intermediate set constructions are finite, i.e., possess a bijection to a finite segment of the natural numbers. ∎

5 Discussion

Related work. The study of self-stabilizing algorithms was initiated by Dijkstra in 1974 [Dij74]. He presented three algorithms for distributed N-process mutual exclusion arranged in a ring. These algorithms ensured that the system converged to stable behavior with at most one privileged or active process in any state. The study of self-stabilizing algorithms was dormant for a while following Dijkstra's work but there has been a recent flurry of activity in the subject. Schneider [Sch93] presents a survey of the issues in the design of self-stabilizing algorithms. Arora and Gouda [AG93] describe a uniform formal framework for verifying fault-tolerance and self-stability. Varghese [Var92] presents various systematic techniques for designing fault-tolerant algorithms including one based on the algorithm studied in this paper. He also provides an informal proof sketch for the correctness of this algorithm that is loosely similar to the one described here.

Prasetya [Pra97] verifies a self-stabilizing minimum-cost routing algorithm in a variant of the UNITY logic [CM88] that is formalized in the HOL proof checking system [GM93]. He presents an elegant development of the theory needed to verify this algorithm but reports a prohibitively high level of verification effort. His verification needed 8900 lines of specification and proof for the basic theories, 9300 lines for formalizing various aspects of the UNITY logic, and 5500 lines for the verification itself. Even though his verification takes some implementation details such as the communication mechanisms into account, the algorithm he verifies is roughly comparable in complexity to the one verified here. Since Prasetya's verification differs from ours in style and content, a sensible comparison of the two proof efforts is not viable. However, in the following discussion, we would like to dispel the impression that the mechanical verification of distributed algorithms, particularly self-stabilizing ones, necessarily requires an unreasonable amount of time and effort.

The challenge of formalization. Having struggled with the formalization of this self-stabilization problem, we certainly do not feel that formalization is a trivial matter. It took us quite a bit of effort to resolve even simple questions such as how to state the pigeonhole principle in its most relevant and usable form, how to formulate the measure function, and when to use measure induction rather than ordinary induction. The proof effort is extremely sensitive to the exact form of the formalization. It took us several false starts and not-so-near misses before we finally arrived at the formalization presented here. The construction of informal proof outline given by the claims in Section 2 was guided by insights gained from the failed attempts but was written without the aid of mechanization.

The final formalization does not dilute the intuitive clarity of the informal presentation. We have given a detailed informal outline of the formal steps in the mechanical verification. Nothing in this outline runs counter to informal intuitions, and indeed the formal proof is a faithful elaboration of these informal intuitions. Several serious gaps in the informal proof sketch were addressed in the formalization without loss of intuitive clarity. It is, of course, quite easy to construct informal or formal definitions and proofs that are completely obscure. However, it is not the case that the clarity of an informal presentation must be lost in formalization.

The formal script is not unreasonably long. The entire specification is fewer than 200 lines including generous amounts of white space and blank lines. The proof script consisting of all of the interactive commands for all of the proofs, including those of the type-correctness proof obligations that were generated and proved automatically, is fewer than 600 lines when pretty-printed with lots of white space and blank lines. This level of conciseness and faithfulness to the informal outline would be impossible without the use of the automation that is available in PVS in terms of typechecking, proof obligation generation, rewriting, decision procedures, and proof strategies. Even this proof script could be made more robust and concise through modest improvements to the automation that is available in PVS.

The mechanization was first fully completed with about four to six days of effort, and another two days were spent streamlining some of the proofs. Automated tools provide quite a bit of help in getting the details right by highlighting both syntactic and semantic errors. In the case of PVS, we use it to actively explore and discover proofs and not merely check existing proofs. This activity is quite enjoyable. The parts that are routine and mechanical are usually handled by means of the automation. The construction of a crisp and taut formal argument requires a number of mental leaps that are essentially similar to the creative effort needed to construct a convincing informal proof. In making these leaps,

the mechanization helps construct explanations why some of these leaps are problematic and how they can be refined.

The main conclusion, one that has been observed earlier by de Bruijn [dB80] and others, is that the machine-checked proofs are of manageable size and linearly track the informal argument, and with modern automation, the expansion factor can actually be made quite small.

The challenge of mechanical verification. There is a widespread belief that mechanized deductive verification is too hard. Even those who advocate formal techniques are skeptical about the value of mechanical verification. The primary objection is that the achievable level of automation is inadequate for productive proof construction. In the case of the self-stabilization proof, even with the extensive automated support in PVS, we were repeatedly confronted with subgoals that were obvious but where the automation in PVS was unable to expand just the right definitions or supply the right instantiations for quantified variables. Through experiments such as these, we are constantly learning about the problems and drawbacks of the current automated support. There is no reason why the automation cannot be improved to a point where most obvious subgoals are indeed proved automatically.

The existing state of automated support is not an indication of what is in fact possible. To quote Dana Scott, *"formalization is an experimental science"* and the same holds for mechanization. Current automated tools are already being used quite effectively by a large body of users. The progress so far in understanding the tradeoffs between automation and interaction has been quite encouraging.

Is self-stability hard to verify? Prasetya [Pra97] claims the verification of self-stability to be especially hard. We can only speak of this particular experience. The main challenge for us was in coming up with a clean and crisp informal proof sketch. The mechanization based on this proof sketch was certainly not a mere formality, but there is no reason to believe that there is anything especially hard about deductively verifying self-stability. Self-stability is a global progress property of a system and requires reasoning about global progress measures which makes it different from the usual local progress properties that are proved for individual processes, but the techniques used are not that dissimilar.

How much of the proof was conventional mathematical reasoning? We would estimate that more than half the verification involved conventional mathematical reasoning about sets, cardinalities, modulo arithmetic, injective maps, pigeonhole principle, and well-founded measure functions. There were only four truly temporal proofs and even some of these involved a fair amount of combinatorial reasoning.

Would a special-purpose temporal formalism have helped? Although we initially felt that this proof would not have derived a significant benefit from a special-purpose temporal formalism [MP92, CM88] because large parts of the proof involved conventional mathematical reasoning, we did find some situations where a temporal framework would have been handy. After we completed our first pass at the verification, we observed that all the temporal properties following Lemma 4.9 employed instances of a single temporal rule, a generalization of the `WHILE` rule, which allows $\Diamond(p \wedge q)$ to be derived from $\Diamond q$ and $p \wedge \Box(p \wedge \neg q \supset \bigcirc p)$. We have not tried out the proof with this proof rule, but believe that

the ability to identify and employ such powerful and generic proof rules provides sufficient methodological justification for embedding a temporal framework within the general-purpose one used here. Temporal logic can be quite easily embedded in the higher-order logic of PVS.

Could model checking be used? Model checking is often the easiest way to automatically verify a distributed algorithm. PVS has support for model checking but we were unable to reduce the problem here to one that could be model checked. While there are some finite-state techniques for verifying parametric systems of the form shown here [CGL94, WL89], they do not appear to be useful for this example. We were also unable to reduce the other two protocols presented by Dijkstra [Dij74] to model checkable form even though these are systems composed only of finite-state processes.

We pose this as an open problem: *Find a way to combine deduction and model checking to verify this and other self-stabilization properties/algorithms, or show that such a reduction is impossible?*

6 Conclusion

We have presented a proof of one of Dijkstra's self-stabilizing mutual exclusion algorithms that has been verified using PVS. We started with an informal proof sketch and constructed a formalization and mechanical verification that preserves the structure and enhances the clarity of the original proof sketch. This serves as a useful existence proof that the challenges of formalization and mechanization are not unreasonably daunting. Dijkstra's algorithm is particularly interesting since its justification employs a nontrivial amount of combinatorial mathematics. This kind of mathematics has typically been thought to be easier to present informally than formally. We have shown that this is not the case and that the formalization can closely follow the informal development. We have also shown exactly where the formal development loses succinctness when compared with the informal one. Such an analysis is useful in developing automated decision procedures and strategies that can more effectively close the informal/formal gap.

Acknowledgment. This work was supported by the Air Force Office of Scientific Research under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the National Science Foundation, or the U.S. Government. We are grateful to John Rushby for his encouragement, advice, and careful reading of drafts of this paper, to Sam Owre for help with PVS and LaTeX, and to Nikolaj Bjørner and the anonymous reviewers for their insightful comments and suggestions.

References

[AG93] Anish Arora and Mohamed Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, November 1993.

[BM84] R. S. Boyer and J. S. Moore. Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly*, 91(3):181–189, 1984.

[CGL94] E.M. Clark, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 589–606. Academic Press, New York, NY, 1980.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[Dij86] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5–6, 1986.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, Cambridge, UK, 1993.

[MP92] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification.* Springer-Verlag, New York, NY, 1992.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Pra97] I. S. W. B. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 399–415, Enschede, The Netherlands, April 1997. Springer-Verlag.

[Sch93] Marco Schneider. Self stabilization. *ACM Computing Surveys*, 25(1):45–67, March 1993.

[Var92] George Varghese. *Self-Stabilization by Local Checking and Correction.* PhD thesis, MIT, 1992.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, number 407 in Lecture Notes in Computer Science, pages 68–80, Grenoble, France, 1989. Springer Verlag.

Shaz Qadeer is a graduate student in Electrical Engineering and Computer Sciences at the University of California at Berkeley. His research interests are formal methods applied to verification and synthesis of concurrent hardware and software systems. He graduated with a B.Tech. in Electrical Engineering and received the President's Gold Medal from the Indian Institute of Technology, Kanpur in 1994. He received an M.S. in Electrical Engineering and Computer Sciences from the University of California at Berkeley in 1997.

Natarajan Shankar is a computer scientist at the SRI International computer science laboratory in Menlo Park. His main research interests are in formal methods and automated deduction. He graduated with a B.Tech. in Electrical Engineering from IIT Madras in 1980, and a Ph.D. in Computer Science from the University of Texas at Austin in 1986.

A Case-Study in Component-Based Mechanical Verification of Fault-Tolerant Programs[1]

To appear in Fourth Workshop on Self-Stabilization (WSS'99) Austin, Texas, USA

Sandeep S. Kulkarni
Department of Computer and Information Science
The Ohio State University
Columbus Ohio 43210 USA

John Rushby Natarajan Shankar
Computer Science Laboratory
SRI International
Menlo Park CA 94025
USA

Abstract

In this paper, we present a case study to demonstrate that the decomposition of a fault-tolerant program into its components is useful in its mechanical verification. More specifically, we discuss our experience in using the theorem prover PVS to verify Dijkstra's token ring program in a component-based manner. We also demonstrate the advantages of component based mechanical verification.

Keywords : Component-based verification, Fault-tolerance, Program decomposition, Mechanical verification, Self-stabilization

1 Introduction

In this paper, we argue that the decomposition of a fault-tolerant program into its components is beneficial in its mechanical verification, and that such a decomposition admits reuse of the proofs for other fault-tolerant programs as well as the variations of the given fault-tolerant program.

Arora and Kulkarni [3] have shown that a fault-tolerant program can be decomposed into a fault-intolerant program and a set of 'tolerance'-components, namely *detectors* and *correctors*. Intuitively, a detector is a component that 'detects' whether a given predicate is true in the program state, and it is used for ensuring that the program satisfies its safety specification in the presence of faults. Likewise, a corrector is a component that 'corrects' the system to a state where the given state predicate is true, and it is used for ensuring that the program eventually recovers to a state from where its specification is satisfied.

For example, a fail-safe program, which satisfies only its safety specification in the presence of faults, can be decomposed into a fault-intolerant program and detector(s). Likewise, a self-stabilizing program, which guarantees recovery to a state from where its specification is satisfied, can be decomposed into a fault-intolerant program and corrector(s).

Decomposition of a fault-tolerant program permits the verification of a given property by focusing on the component that is responsible for satisfying it. For example, if we need to show that a program eventually recovers to a state from where it satisfies its specification, we should focus on its corrector components. Likewise, if we are interested in showing that the program satisfies its specification in the absence of faults, we should focus on the corresponding fault-intolerant program. Of course, we will have to show that other components of the program do not interfere with the component of interest. But this proof is typically simpler than the proof required to show that the overall program satisfies the given property. Moreover, if we change some components used in that program, the proofs of other components are not affected. Thus, it is possible for a small change in the program to lead to a small change in the proof.

With the motivation of developing a systematic approach for mechanical verification using program decomposition, we are implementing the theory of detectors and correctors into the theorem prover PVS [11][2]. In this paper, we present a proof of one of Dijkstra's token ring program [5] that has been proved using this theory. Previously, Qadeer and Shankar [13] have verified this token ring program using PVS. While their proof is impressive, it is very specific to one program and, hence, much of their proof-technique cannot be reused to prove other fault-tolerant programs. Moreover, since they focus on the entire program, instead of its components, their proof is more complex than it needs to be. We use this case-study to illustrate how the decomposition of the program into its components can help in making the proofs simple and reusable.

Being self-stabilizing, Dijkstra's program can be decomposed into a fault-intolerant program and a corrector. The fault-intolerant program circulates the token along an initialized ring in the absence of faults. On the other hand, if faults perturb the program from its ideal states, the corrector restores the fault-intolerant program back to some ideal state, from where it continues to circulate the token. This program is self-stabilizing in that even if the faults perturb the program to an arbitrary state, the corrector restores it to

[1]Email: kulkarni@cis.ohio-state.edu, rushby@csl.sri.com, shankar@csl.sri.com Web: `http://www.cis.ohio-state.edu/~kulkarni`, `http://www.csl.sri.com/~rushby`, `http://www.csl.sri.com/~shankar`

This work was partially supported by National Science Foundation grant CCR-9509931 and by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

[2]The URL `http://www.cis.ohio-state.edu/~kulkarni/pvs/` contains the PVS specification and proofs.

an ideal state.

In Dijkstra's token ring program, processes $0..N$, $N \geq 1$, are organized in a ring. Each process j maintains a counter $x.j$, $0 \leq x.j < M$ for some $M > 1$. A non-zero process j has a token iff $x.j$ differs from $x.(j-1)$, and process 0 has a token iff $x.0$ is the same as $x.N$. If process j has a token then it passes it to process $j+1 \ mod \ N+1$ by setting $x.j$ to $x.(j-1)$, and if process 0 has a token then it passes it to process 1 by incrementing $x.0$. For any M, $M > 1$, the program guarantees that in the absence of faults there will be exactly one token that is being circulated in the ring. If $M \geq N+1$, the program guarantees that starting from any arbitrary state, the program will reach a state where there is exactly one token which is circulated along the ring.

To decompose Dijkstra's program into a fault-intolerant program and a corrector, we first consider the following question: If we are only interested in a token circulation along an initialized ring, how can the token ring program be simplified? The answer to this question identifies the fault-intolerant program. Next we ask the question about fault-tolerance: what are the ideal states of the resulting fault-intolerant program, and how can it be recovered to these ideal states if the faults perturb it? The answer to this question identifies the corrector. Then, we show how the fault-intolerant program and the corrector can be independently verified in PVS and how they can be shown to be interference-free.

The rest of the paper is organized as follows: In Section 2, we present Dijkstra's token ring program and its decomposition into a fault-intolerant program and a corrector. In Section 3, we show how the token ring program is modeled in PVS. In Section 4 and 5, we present the correctness proof for the fault-intolerant program and the corrector respectively. In Section 6, we show that the corrector and the fault-intolerant program do not interfere with each other. Finally, in Section 7, we discuss the advantages of component-based verification over non-component-based verification, and present concluding remarks in Section 8.

2 The Token Ring Program and its Decomposition

In this section, we first present the decomposition of Dijkstra's token ring program into a fault-intolerant program and a corrector. Then, we argue that they work in isolation and that they do not interfere with each other. We use the same arguments for mechanical verification in Sections 4, 5 and 6.

Fault-intolerant program. If we are not interested in fault-tolerance, a token ring program can be designed by maintaining a variable $x.j$ (in the range 0..(M-1), where $M > 1$) as follows: Each non-zero process j checks whether $x.j$ is different from $x.(j-1)$. If this condition is true then $x.j$ is set to $x.(j-1)$. Process 0 checks whether $x.0$ is the same as $x.N$. If this condition is true, process 0 increments $x.0$. Thus, the actions of the fault-intolerant program are as follows:

$$\begin{array}{lll} j \neq 0 \ \wedge \ x.j \neq x.(j-1) & \longrightarrow & x.j := x.(j-1) \\ x.0 = x.N & \longrightarrow & x.0 := x.0 + 1 \end{array}$$

The invariant of this program is S, where

$$\begin{array}{rl} S \ = \ (\exists j, v : 0 \leq j \leq N, 0 \leq v < M \ : & \\ \qquad (\forall k : 0 \leq k < j : x.k = v) \ \wedge & \\ \qquad (\forall k : j \leq k \leq N : x.k = v - 1 \ mod \ M)) & \end{array}$$

The invariant S characterizes the states where there exists a process j such that the x values of processes $0..(j-1)$ are equal to v, and the x values of processes $j..N$ are equal to $v-1 \ mod \ M$. Thus, process j has the unique token, and only the action at j is enabled in that state. The execution of this action results in a state where process $j+1 \ mod \ N+1$ has the token. Thus, starting from a state where S is true, the fault-intolerant program circulates a unique token along the ring.

Corrector. If the faults perturb the x values maintained at the processes, we need to recover the program to a state where S holds in order to ensure that the token circulation is re-established. This can be achieved by the corrector that lets each non-zero process copy the x value of its predecessor. Thus, the actions of the corrector are as follows:

$$j \neq 0 \ \wedge \ x.j \neq x.(j-1) \quad \longrightarrow \quad x.j := x.(j-1)$$

Observe that if the corrector actions execute in isolation, a state is reached where all x values are same, and at that state S is true. Also, if the corrector executes in any state where S is true, S continues to be true in the resulting state.

Note that although the actions of the fault-tolerant program and that of the fault-intolerant program are the same, when dealing with the fault-intolerant program we can assume that the invariant S is true. In this sense, the fault-intolerant program is simpler than the fault-tolerant program. Of course, the actions of the corrector are a subset of the fault-tolerant program and, hence, the corrector is simpler than the fault-tolerant program.

Interference-freedom between the fault-intolerant program and the corrector. Since the corrector is a subset of the fault-intolerant program, it is trivial that the corrector does not interfere with the fault-intolerant program. Likewise, the actions of the fault-intolerant program at non-zero processes are a subset of the corrector and, hence, do not interfere with the corrector. Thus, we only need to show that the action at process 0 does not interfere with the corrector. We prove the interference-freedom as follows:

1. For process 0 to interfere with the corrector, it must execute infinitely often. Otherwise, after 0 stops executing, convergence to S will be achieved.

2. If the action at process 0 executes infinitely often, $x.0$ will take all possible values in the range $0..(M-1)$.

3. If the domain of x is large enough, specifically $M \geq N+1$, then in the initial state, there must be a value in the range $0..(M-1)$ which is not present at any non-zero process.

4. From 2 and 3, it follows that eventually $x.0$ will obtain a value missing in the initial state.

5. After $x.0$ is equal to this missing value, process j will obtain this missing value only after processes $0..(j-1)$ obtain this missing value. Thus, when process 0 executes next (from 1, we know that process 0 will execute next), all processes will have the same x value. Thus, a state where S is true is reached.

3 Modeling of the Token Ring in PVS

In this section, we discuss how we modeled Dijkstra's token ring program in PVS. More specifically, we first define program independent concepts such as states, state predicates, actions, program compositions, etc. Then, we define the actions of the token ring program and its invariant.

State. The state of the program consists of the x values at processes $0..N$, each x value is in the range $0..(M-1)$.

Trace. A trace is an infinite sequence of states. If seq is a trace and i is a natural number then $seq(i)$ denotes the i^{th} element in seq.

Assertion. An assertion is a predicate over states. If P is an assertion and s is a state then $P(s)$ denotes whether P is true in state s.

Action. An action is a relation over states. If A is an action and $s1, s2$ are states then $A(s1, s2)$ denotes whether state $s2$ can be reached by executing A in state $s1$.

Property. A property is a predicate over traces. If R is a property and seq is a trace then $R(seq)$ denotes whether the property R is true of seq.

Notation. Henceforth, we use p and q to denote programs; $s, s0, s1$ and $s2$ to denote program states; seq to denote a trace; S and T to denote assertions; R to denote a property; m, n to denote natural numbers; j, k to denote processes; and $v, v1, v2$ to denote the x values at processes. Moreover, given a state s, $x(s)(j)$ denotes the value of $x.j$ in state s.

Program compositions. In the base case, a program is just a single action. The parallel composition of programs p and q, denoted as $p[]q$, is a program consisting of the actions of p and the actions of q. (While we have defined other program compositions used for fault-tolerant programs, we omit them here as they are not used in Dijkstra's token ring program.)

CanExecute. Program p can execute in state $s1$ iff there exists a state $s2$ such that $p(s1, s2)$ is true.

$$CanExecute(p)(s1) \;=\; (\exists s2 :: p(s1, s2))$$

Next. The predicate $Next(p)(s1, s2)$ denotes whether state $s2$ can be reached by execution of some action of p. If no action of p is enabled in state $s1$ then $Next(p)(s1, s2)$ is true iff $s1 = s2$.

$$\begin{aligned} Next(p)(s1, s2) \;=\; & (CanExecute(p)(s1) \;\wedge\; p(s1, s2)) \\ \vee\; & (\neg CanExecute(p)(s1) \;\wedge\; s1 = s2) \end{aligned}$$

Computation. A computation of a program p is a trace $s_0, s_1, ...$ such that for each n, $Next(p)(s_n, s_{n+1})$ is true. Thus, the predicate characterizing 'seq is a computation of program p' is represented as follows:

$$run(p)(seq) \;=\; \forall n :: Next(p)(seq(n), seq(n+1))$$

Satisfies. Program p satisfies a property R iff for every trace that is a computation of p, $R(seq)$ is true. Thus, $satisfies(p)(R)$ is defined as follows:

$$satisfies(p)(R) \;=\; \forall seq :: run(p)(seq) \Rightarrow R(seq)$$

We use two types of properties in the proof of Dijkstra's token ring program, closure and convergence.

Closure. The property $closed(S)$ is the set of all traces $s_0, s_1, ...$ where for each $n, n \geq 0$, if S is true at s_n then S is true s_{n+1}. Thus, $closed(S)$ is defined as follows:

$$closed(S)(seq) \;=\; \forall n :: S(seq(n)) \Rightarrow S(seq(n+1)))$$

Convergence. The property $converges(T, S)$ is the set of all traces $s_0, s_1, ...$ where $closed(S)$ and $closed(T)$ are true, and if there exists $n, n \geq 0$, for which T is true at s_n then there exists m, $m \geq n$, for which S is true at s_m. Thus, $converges(T, S)$ is defined as follows:

$$\begin{aligned} & converges(T, S)(seq) \;= \\ & \quad closed(T)(seq) \;\wedge\; closed(S)(seq) \;\wedge \\ & \quad \forall n :: T(seq(n)) \;\Rightarrow\; (\exists m : m \geq n : S(seq(m))) \end{aligned}$$

Num_steps. Given an action ac, a trace seq, and a natural number n, the number of times action ac is executed until the n^{th} state is defined as follows:

$$num\_steps(ac)(seq, n) \;= \begin{cases} 0 & \text{if } n = 0 \\ num\_steps(ac)(seq, n-1) + 1 & \text{if } ac(seq(n-1), seq(n)) \\ num\_steps(ac)(seq, n-1) & \text{otherwise} \end{cases}$$

Corrector. The corrector action at a non-zero process j is executed only in states where $x.j$ differs from $x.(j-1)$. The execution of this action results in a state where $x.j$ has the same value as that of $x.(j-1)$ and the other x values remain unchanged. Thus, corrector action at j is defined as follows:

$$\begin{aligned} corr(j)(s0, s1) \;=\; & x(s0)(j) \neq x(s0)(j-1) \;\wedge \\ & x(s1)(j) = x(s0)(j-1) \;\wedge \\ & \forall k : k \neq j : x(s1)(k) = x(s0)(k) \end{aligned}$$

The corrector consists of the actions at all non-zero processes. We, therefore, use parallel composition of $corr(j)$, $0 < j \leq N$, to define the corrector, $corr\_prog$, as follows :

$$corr\_prog \;=\; ([]j : j \neq 0 : corr(j))$$

Action at process 0. The action at process 0 is executed only in states where $x.0$ is the same as $x.N$. The execution of this action results in a state where the value of $x.0$ is one greater than its initial value (in mod M arithmetic) and the other x values remain unchanged. Thus, the action at process 0 is defined as follows:

$$\begin{aligned} & action\_zero(s0, s1) \;= \\ & \quad x(s0)(0) = x(s0)(N) \;\wedge \\ & \quad x(s1)(0) = x(s0)(0) + 1 \; mod \; M \;\wedge \\ & \quad \forall j : j \neq 0 : x(s1)(j) = x(s0)(j) \end{aligned}$$

Note that the fault-intolerant program consists of the parallel composition of the action at process 0 and the corrector. Thus, the fault-intolerant program is $action\_zero[]corr\_prog$.

j has a token. We define the predicate, 'j has a token in state s' as follows:

$$\begin{aligned} token(s)(j) \;=\; & (j = 0 \;\wedge\; x(s)(0) = x(s)(N)) \\ \vee\; & (j \neq 0 \;\wedge\; x(s)(j) \neq x(s)(j-1)) \end{aligned}$$

Invariant of the fault-intolerant program. Finally, we define the invariant of the fault-intolerant program, $corr\_pred$, as follows:

$$corr\_pred(s) \;=\; (\exists j, v :: \; \forall k : k<j : x(s)(k) = v \;\wedge\; \forall k : k \geq j : x(s)(k) = v-1 \bmod M)$$

Remark. Although in this presentation, we have given a specific instantiation for the program state, it is initially defined as an uninterpreted type, and then instantiated suitably for the token ring program. This allows program independent concepts such as traces, assertions to be reused for different programs.

4 Verification of the Fault-Intolerant Program

To prove the correctness of the fault-intolerant program, we need to show (1) *corr_pred* is closed in the fault-intolerant program, and (2) if the token is at process j and an action of the fault-intolerant program is executed then the token is at process $j+1 \bmod N+1$ in the resulting state.

In Theorem 4.3, we show that *corr_pred* is closed in the fault-intolerant program. In this proof, we use Lemmas 4.1 and 4.2 which show that *corr_pred* is closed in the action of process 0 and the actions of non-zero process respectively. Finally, we show the token circulation property in Theorem 4.5.

Lemma 4.1 In the computation of *action_zero* alone, *corr_pred* is closed. Formally,

$$satisfies(action\_zero)(closed(corr\_pred))$$

Proof. After eliminating the quantifiers and expanding the definitions, we need to show that if *corr_pred* is true in the n^{th} state of the computation then it is true in $(n+1)^{th}$ state of that computation. To this end, we first do a case split on the process that has the token in the n^{th} state: In the case, where process 0 has the token, i.e., the x values of processes $0..N$ are $v-1 \bmod M$ for some v, we show that execution of *action_zero* results in a state where the x values of $1..N$ remain $v-1 \bmod M$ and the x value of 0 is v, i.e., *corr_pred* is true. In the case, where process j, $j \neq 0$, has the token, we show that $x.0$ is v and $x.N$ is $v-1 \bmod M$ for some v and, hence, *action_zero* is disabled, i.e., the $(n+1)^{th}$ state is identical to the n^{th} state and, hence, *corr_pred* is true in $(n+1)^{th}$ state. □

Lemma 4.2 In the computation of the action at a non-zero process, *corr_pred* is closed. Formally,

$$\forall j :: satisfies(corr(j))(closed(corr\_pred))$$

Proof. The proof of this lemma is similar to that of Lemma 4.1. We show that if process j has the token then in the resulting state process $j+1 \bmod N+1$ has the token, and if any other process has the token, the execution results in a stuttering. In either case, *corr_pred* is true. □

Theorem 4.3 In the computation of the fault-intolerant program, *corr_pred* is closed. Formally,

$$satisfies(action\_zero[]corr\_prog)(closed(corr\_pred))$$

Proof. This lemma is proved by using Lemmas 4.1 and 4.2 and the following property about parallel composition: if an assertion S is closed in programs p and q then it is closed in $p[]q$. □

Lemma 4.4. At least one action of the fault-intolerant program is enabled in any program state. Formally,

$$\forall s :: CanExecute(action\_zero[]corr\_prog)(s)$$

Proof. We prove this lemma by first doing a case-split on whether all x values are equal. If all x values are equal, it follows that $x.0 = x.N$ and, hence, *action_zero* is enabled. If all x values are not equal, we induct on the processes to find the first process, say j, such that $x.j$ differs from $x.0$. Since $x.(j-1) = x.0$ and $x.j \neq x.0$, it follows that process j is enabled. □

Theorem 4.5 Starting from a state where *corr_pred* is true, if the token is at process j then the execution of an action of the fault-intolerant program results in a state where the token is at process $j+1 \bmod N+1$. Formally,

$$\begin{array}{l} \forall j :: \quad token(s1)(j) \\ \qquad \wedge\ corr\_pred(s1) \\ \qquad \wedge\ Next(action\_zero[]corr\_prog)(s1, s2) \\ \quad \Rightarrow \\ \qquad (j \neq N \Rightarrow token(s2)(j+1)) \\ \qquad \wedge\ (j = N \Rightarrow token(s2)(0)) \end{array}$$

Proof. Lemma 4.4 shows that execution of the fault-tolerant program does not result in stuttering. We then show that if process j has the token no other process is enabled. Finally, we show, as in Lemmas 4.1 and 4.2, that the execution of the action at j results in a state where $j+1 \bmod N+1$ has the token. □

5 Verification of the Corrector

To prove that *corr_prog* satisfies its specification, we need to show (1) *corr_pred* is closed in *corr_prog*, and (2) starting from any state, in the execution of *corr_prog* alone a state is reached where *corr_pred* is true. Note that (1) follows from Lemma 4.2. In this section, we prove that the corrector satisfies property (2) based on the following observation:

Observation. If only the actions at processes $j..N$ execute in a computation then eventually the x values of processes $j-1..N$ will be identical and the actions of processes $j..N$ will be disabled. □

If $j = 1$, the actions at processes $j..N$ are the same as the actions of the corrector and, hence, in the execution of the corrector, eventually the x values of all processes will be identical. Thus, convergence to *corr_pred* is achieved.

In order to obtain the proof of the above observation in PVS, we first define the program consisting of actions of processes $j..N$ and an assertion characterizing the states where the x values of processes $j..N$ are equal. Then, we provide a proof of the above observation in Lemma 5.2. Finally, we prove the convergence property in Theorem 5.3.

We define $corr\_above(j)$, and $same\_as\_N(j)$ as follows:

Definition $corr\_above(j)$. For any j, $j \neq 0$, $corr\_above(j)$ is the program consisting of the actions at processes $j..N$. Formally,

$$corr\_above(j) = ([]k : j \leq k \leq N : corr(k)) \qquad \square$$

Definition $same\_as\_N(j)$. For any j, $same\_as\_N(j)$ is an assertion which is true in state s iff the x values of processes $j..N$ are identical in state s. Formally,

$same\_as\_N(j)(s) =$
$\forall k : j \leq k \leq N : x(s)(k) = x(s)(N)$ □

Lemma 5.1 If $x.j$ is the same as $x.(j-1)$ in some state in the computation of $corr\_above(j)$, then this condition continues to be true in the rest of the computation. Formally,

$$\forall seq, j, n : j \neq 0 : \; run(corr\_above(j))(seq) \wedge x(seq(n))(j-1) = x(seq(n))(j)) \Rightarrow \forall m : m \geq n : x(seq(m))(j-1) = x(seq(m))(j)$$

□

Lemma 5.2 In the computation of the corrector actions at processes $j..N$, a state is reached where the x values of processes $(j-1)..N$ are identical. Formally,

$$\forall seq, j :: \; run(corr\_above(j))(seq) \Rightarrow \exists n :: same\_as\_N(j-1)(seq(n))$$

Proof. We prove this lemma by measure-induction on the j, where the measure used is $N-j$. In the base case, $j=N$, we do a case split on whether $corr(N)$ is enabled in the initial state: If $corr(N)$ is enabled, we show that in the successor state, $same\_as\_N(N-1)$ is true. If $corr(N)$ is disabled, we show that in the initial state $same\_as\_N(N-1)$ is true.

In the induction case, we do a case-split on whether the action at process j executes in the computation of $corr\_above(j)$. If j executes in the n^{th} state, we show that the suffix of the computation from the $(n+1)^{th}$ state is a computation of $corr\_above(j+1)$. Therefore, there exists a state, say s, where $same\_as\_N(j)$ is true. Moreover, since the values of $x.j$ and $x.(j-1)$ are equal in the $(n+1)^{th}$ state, by Lemma 5.1, it follows that the values of $x.j$ and $x.(j-1)$ are equal in state s. Thus, $same\_as\_N(j-1)$ is true in state s.

If j never executes in the computation of $corr\_above(j)$, we show that that computation is also a computation of $corr\_above(j+1)$. Therefore, there exists a state, say s, in this computation where $same\_as\_N(j)$ is true. Thus, the actions of $j+1..N$ are disabled in state s. Since the program tries to execute an action unless all its actions are disabled, it follows that the action at j must also be disabled. It follows that $same\_as\_N(j-1)$ is true in s. □

Theorem 5.3 The computation of the corrector eventually converges to $corr\_pred$. Formally,

$$satisfies(corr\_prog)(converges(true, corr\_pred))$$

Proof. We prove this lemma by instantiating $j=1$ in Lemma 5.2. From Lemma 5.2, it follows that in the computation of the corrector alone, eventually a state is reached where all x values are identical and, hence, $corr\_pred$ is true in that state. Moreover, the closure of $corr\_pred$ follows from Lemma 4.2. □

6 Interference-Freedom Between the Corrector and the Fault-Intolerant Program

After we showed that the fault-intolerant program and the corrector satisfy their specification in isolation, we proceed to show that they do not interfere with each other. As mentioned in Section 2, towards this end, we show that the action at process 0 does not interfere with the corrector. Our proof follows the outline discussed in Section 2. More specifically, in Lemma 6.1, we show that process 0 executes infinitely often. Then, in Lemma 6.4, we show that there exists a value that is different from the x values of all non-zero processes. Subsequently, in Lemma 6.6, we show that eventually process 0 gets this missing value, and in Theorem 6.8, we conclude that the action at process 0 does not interfere with the corrector.

Lemma 6.1 In the computation of the corrector and the action at process 0, either process 0 executes infinitely often or a state is reached where $corr\_pred$ is true. Formally,

$$\forall seq, n :: \; run(action\_zero [] corr\_prog)(seq) \Rightarrow \forall m :: (\exists n : n \geq m : action\_zero(seq(n), seq(n+1))) \vee \exists m :: corr\_pred(seq(m))$$

Proof. Note that process 0 executes infinitely often iff given any number m, it executes in the n^{th} state for some $n \geq m$. Thus, to prove this lemma we need to show that either (1) there exists a number n, $n \geq m$, such that $action\_zero$ executes in the n^{th} state, or (2) there exists a state where $corr\_pred$ is true. We prove this lemma by a case-split on whether the suffix of the computation starting from m^{th} state is a computation of $corr\_prog$. If that suffix is a computation of $corr\_prog$, by Theorem 5.3, it is straightforward to show that (2) is true. If the suffix is not a computation of $corr\_prog$, there exists a number n, $n \geq m$, such that the $(n+1)^{th}$ state is not obtained by executing $corr\_prog$ in the n^{th} state. Since in the n^{th} state either $action\_zero$ executes or $corr\_prog$ executes, it follows that in the n^{th} state $action\_zero$ executes, i.e., (1) is true. □

Lemma 6.2 In the computation of the corrector and the action at process 0, the value of $x.0$ in the n^{th} state of the computation is equal to the sum of the initial value of $x.0$ and the number of steps taken by process 0. Formally,

$$\forall seq :: \; run(action\_zero [] corr\_prog)(seq) \Rightarrow x(seq(n))(0) = (x(seq(0))(0) + num\_steps(action\_zero)(seq, n)) \; mod \; M$$

Proof. We prove this lemma by induction on the length of the computation. In the initial state, this condition is trivially satisfied. In the induction case, we do a case-split on whether process 0 executes or whether $corr\_prog$ executes. In each case, the proof is straightforward. □

Lemma 6.3 In the computation of the corrector and the action at process 0, either $x.0$ takes on all possible values in the range $0..(M-1)$ or a state is reached where $corr\_pred$ is true. Formally,

$$\forall seq :: run(action\_zero [] corr\_prog)(seq) \Rightarrow \forall v : 0 \leq v < M : (\exists n :: x(seq(n))(0) = v) \vee \exists n :: corr\_pred(seq(n))$$

Proof. We prove this lemma by using Lemmas 6.1 and 6.2 . In Lemma 6.2, if the value of $x.0$ in the initial state is $v0$ then after process 0 executes $(v - v0) \; mod \; M$ steps, the value of $x.0$ will be v. By Lemma 6.1, either process 0 executes

infinitely often or a state is reached where *corr_pred* is true. In the former case, we know that process 0 executes $(v - v0)\ mod\ M$ times and, hence, the value of $x.0$ is eventually v. In the latter case, the lemma is trivially true. □

Lemma 6.4 If $M \geq N+1$, then in any state there exists a value, say v, in the range $0..(M-1)$ such that the x values of all non-zero processes are different from v. Formally,

$$\forall s :: (\exists v : 0 \leq v < M : (\forall j : j \neq 0 : x(s)(j) \neq v))$$

Proof. Note that this lemma essentially states the pigeonhole principle: There are at most N distinct x values of non-zero processes and, hence, if $M \geq N+1$, there must exist a value that is different from the x values of all non-zero processes. We prove this lemma in the following steps:

(1) $|\{x.j : j \neq 0\}|$ is at most N,

(2) $(|\{v : 0 \leq v < M\} - \{x.j : j \neq 0\}|)$ is non-zero if $M \geq N+1$.

We use the set library in PVS in our proof of (1) and (2). This library defines various operations with sets such as union, intersection, difference, cardinality, etc, and provides some standard lemmas about them.

Given a state s, we define the set of x values of non-zero processes upto j, $nonz\_set\_upto(s)(j)$ as follows:

$$nonz\_set\_upto(s)(j) = \begin{cases} \{\} & \text{if } j = 0 \\ nonz\_set\_upto(s)(j-1) \cup \{x(s)(j)\} & \text{otherwise} \end{cases}$$

By induction on j, we then prove that the cardinality of $nonz\_set\_upto(s)(j)$ is atmost j: The base case, $j = 0$, is trivial since $nonz\_set\_upto(s)(0)$ is the empty set. For the induction case, we use the fact that $nonz\_set\_upto(j+1) = nonz\_set\_upto(j) \cup \{j+1\}$ and that the cardinality of the union is no greater than the sum of cardinalities. Now, observe that (1) is trivially true if we instantiate $j = N$.

To prove (2), we use the fact that the for any two sets X and Y, $|X - Y| \geq |X| - |Y|$. Letting X be the set $0..(M-1)$, and Y be the set x values of non-zero processes, we show that in any state there exists a missing value, i.e., a value that is different from the x values of all non-zero processes. □

To identify some missing value in state s, we define a constant $missing(s)$ which denotes some arbitrary value that is missing in state s .

Definition. Given a state s, $missing(s)$ is some arbitrary value in the set $(\{v : 0 \leq v < M\} - \{x.j : j \neq 0\})$ □

Remark. Note that the definition of $missing(s)$ is sensible only if $M \geq N+1$. Thus, all theorems that use this definition rely on this assumption. For brevity, however, we will not explicitly specify this assumption in the subsequent theorems. In PVS, we define $M \geq N+1$ as an axiom so that it can be omitted in the statement of the theorems.

Lemma 6.5 Let s be any state in the computation of the corrector and the action at process 0. The x value of any non-zero process in s is either present in the initial state of that computation or it is generated by process 0 in a state preceding s. Formally,

$$\begin{array}{l} \forall seq, n :: \\ \quad run(action\_zero \| corr\_prog)(seq) \\ \Rightarrow \\ \quad \forall j : j \neq 0 : \quad (\exists k :: x(seq(n))(j) = x(seq(0))(k)) \\ \qquad\qquad \vee\ (\exists m : m < n : \\ \qquad\qquad\quad x(seq(n))(j) = x(seq(m))(0)) \end{array}$$

Proof. We prove this lemma by induction on the length of the computation. The base case, the initial state, is trivial; the x values of all non-zero processes are present in the initial state.

For the inductive case, our proof obligation is that if the x value of a non-zero process is changed in the $(n+1)^{th}$ state then that new value is either present in the initial state or it is generated by process 0 in an earlier state. Towards this end, we first do a case-split on which process executes in the n^{th} step: If process 0 executes in the n^{th} step, the x values of non-zero processes remain unchanged. Thus, the proof obligation is trivially satisfied. If a non-zero process, say j, executes in the n^{th} step, only the value of $x.j$ is changed it is set to $x.(j-1)$. We then do a case-split on whether $j = 1$ or $j \neq 1$. If $j = 1$, we show that the value of $x.j$ in the $(n+1)^{th}$ state is generated by process 0 in the n^{th} state. If $j \neq 1$, by induction on the value of $x.(j-1)$, it follows that the new value of $x.j$ is either present in the initial state or it is generated by process 0 in an earlier state. □

Lemma 6.6 In the computation of the corrector and the action at process 0, a state is reached that satisfies one of the following conditions: (1) $x.0$ is equal to a value missing in the initial state and the x values of non-zero processes are different from $x.0$, or (2) *corr_pred* is true. Formally,

$$\begin{array}{l} \forall seq :: \quad run(action\_zero \| corr\_prog)(seq) \\ \quad \Rightarrow \\ \qquad \exists n :: x(seq(n))(0) = missing(seq(0)) \ \wedge \\ \qquad\quad (\forall j : j \neq 0 : \\ \qquad\qquad x(seq(n))(j) \neq missing(seq(0))) \\ \qquad \vee\ \exists n :: corr\_pred(seq(n)) \end{array}$$

Proof. From Lemma 6.3, in the computation of the corrector and the action at process 0, a state is reached where either $x.0 = missing(seq(0))$ is true or *corr_pred* is true. In the latter case, Lemma 6.6 is trivially satisfied. In the former case, we induct on the length of the computation to show that there exists a state, say s, such that $x.0 = missing(seq(0))$ is true in state s, and $x.0 = missing(seq(0))$ is false in all states preceding s in the computation. We then use Lemma 6.5 to show that in state s, $x.0$ is different from the x values of all non-zero processes. By the construction of s, $x.0$ is never equal to $missing(seq(0))$ in any state preceding s. Moreover, by definition of $missing(seq(0))$, it is not present in the initial state at any non-zero process. Thus, from Lemma 6.5, it follows that in state s, the value of $x.0$ is different from the x values of non-zero processes. □

Lemma 6.7 If the corrector executes starting from a state where $x.0$ differs from the x values of all non-zero processes then in any state of that computation if $x.0$ is the same as $x.j$ then the x values of processes $0..j$ are the same as $x.0$. Formally,

$$\begin{array}{l} \forall seq :: \quad run(corr\_prog)(seq) \\ \qquad \wedge\ (\forall j : j \neq 0 : x(seq(0))(j) \neq x(seq(0))(0)) \\ \quad \Rightarrow \\ \qquad (\forall n, j :: x(seq(n))(j) = x(seq(0))(0) \Rightarrow \\ \qquad\quad (\forall k : 0 \leq k \leq j : \\ \qquad\qquad x(seq(n))(k) = x(seq(0))(0))) \end{array}$$

Proof. We prove this result by induction on the length of the computation as well. In the induction case, let $j1$ be

a process that executes in the n^{th} step, and let $j2$ be any process that satisfies $x.j2 = x.0$ in the $(n+1)^{th}$ state. To prove that in the $(n+1)^{th}$ state the x values of $0..j2$ are the same as $x.0$, we do a case-split on whether $j2 < j1$, $j2 = j1$, or $j2 > j1$.

In the first case, we show that the x values of processes $0..j2$ remain unchanged and, hence, $x.j2 = x.0$ must be true in the n^{th} state. Therefore, in the $(n+1)^{th}$ state of the computation, the x values of processes $0..j2$ are the same as $x.0$.

In the second case, we show that it must be the case that in the n^{th} state, $x.(j2-1)$ is the same as $x.0$. Hence, in the n^{th} state, the x values of $0..(j2-1)$ are the same as $x.0$. Since in the $(n+1)^{th}$ state $x.j2$ is the same as $x.0$ and the x values of processes $0..(j2-1)$ remain unchanged, it follows that in the $(n+1)^{th}$ state the x values of processes $0..j2$ are the same as $x.0$.

In the third case also, $x.j2$ remains unchanged. Thus, in the n^{th} state, the $x.j2 = x.0$ is true. Therefore, in the n^{th} state $x.j = x.0$ and $x.(j1-1) = x.0$ is also true. Thus, the action of process $j1$ is disabled. □

Theorem 6.8 The action at process 0 does not interfere with the corrector, i.e., the computation of $action\_zero \| corr\_prog$ converges to $corr\_pred$. Formally,

$$satisfies(action\_zero \| corr\_prog)$$
$$(converges(true, corr\_pred))$$

Proof. We use Lemmas 6.1, 6.4 and 6.7 to prove the above lemma. From 6.4, a state, say s, is reached where $x.0$ differs from the x values of all non-zero processes. From Lemma 6.1, process 0 executes after s. Until 0 executes for the first time, the corresponding computation is a computation of the corrector. Moreover, when 0 executes $x.0$ is the same as $x.N$. Hence, by Lemma 6.7, the x values of all processes are identical. It follows that when 0 executes for the first time after state s, $corr\_pred$ is true. □

7 Discussion

Related work. Since Dijkstra presented the self-stabilizing token ring program in 1974, it has been proved using various techniques [1, 6, 10, 13, 15]. Of these, the proofs by Qadeer and Shankar [13] and Merz [10] have been verified by a theorem prover. Merz constructs a complicated variant function–consisting of the enabled processes, the distance between the x value of the process 0 and the missing value, etc.—and shows that it decreases in every step. In terms of number of interactions required with the theorem prover, it outperforms the proof presented in this paper as well as that by Qadeer and Shankar. However, these reduced interactions come at a very high cost; the creativity required to find this variant function. Also, that proof is hard to comprehend since it does not match with the intuitive understanding of the token ring program.

Qadeer and Shankar closely follow the proof by Varghese [15], and their proof is simpler than that by Merz. However, since they try to prove the properties of the entire program, some of their proofs are more complex than they need to be. For example, they prove that each process eventually gets the token using the following variant: $p(j) = sum\{k : k$ has a token : $(i-j) \ mod \ N + 1 \}$. One of the reasons they need such a variant function is that they are trying to prove that *starting from an arbitrary state* eventually each process will get the token. However, this property is more general than necessary; one only needs to prove that *after the invariant is established*, eventually each process will get the token. Since we prove the token circulation property only in the invariant states, we do not need such a variant function.

In related work on mechanical verification of self-stabilization, Prasetya [12] has verified a self-stabilizing routing program in a variant of UNITY logic [4] using the theorem prover HOL [7]. He also presents an elegant development of the theory needed in the verification but he seems to require a prohibitively high level of verification effort.

Advantages of component-based mechanical verification. Fault-tolerant programs are often tricky and so need strong assurance; mechanical verification is a very strong form of assurance but previous examples were tours-de-force that required great insight and talent and are not readily transferable to other problems or other people. By way of contrast, our component-based approach is systematic and offers some hope of making these verifications routine. The detector-correctors theory and its application to Dijkstra's token ring program shows that the effort required as well as the amount of invention is reduced. We find that the advantages of component-based mechanical proofs are the same as that of component-based non-mechanical proofs. We discuss some of these advantages below.

Reusability for a variation of the token ring program. The modification of a component in the program preserves the correctness proofs of other components. We find that this property is useful in the mechanical verification of the resulting program as well. For example, observe that if the action at process 0 is changed so that $x.0$ is incremented by k (instead of 1), where k is relatively prime to M, the self-stabilization is preserved. After we proved the correctness of Dijkstra's token ring program, we verified the self-stabilization property of this new program and found that it took approximately 30 additional minutes to obtain the new proof (compared to approximately 4-5 days for the initial proof), and most of the proof was reused.

Reusability of proofs for other fault-tolerant programs. Lemma 6.1 shows that either process 0 executes infinitely often or the correction predicate is established. This proof only depends on the fact that the corrector satisfies its specification in isolation, and not on the actual programs and predicates involved. We, therefore, have extracted a simple interference-freedom lemma that is applicable in other programs. Likewise, Lemma 5.2, only depends upon the ordering between the corrector actions. Such a ordering exists in various programs—including most tree based programs. Therefore, the same proof technique can be used in those programs as well. Also, lemmas that relate to program compositions or interference-freedom techniques such as superposition and eventual termination can be reused in other fault-tolerant programs.

Role of assumptions. Observe that our proof clearly shows the assumption $M \geq N+1$ is not required for the correctness of the fault-intolerant program or the corrector; it is required only to prove that they do not interfere with each other. Thus, if we were to weaken this assumption—say because it is possible to prove stabilization when $M \geq N$—we will need to redo only the proofs that depend on this assumption, namely Lemmas 6.4, 6.6 and 6.8. Likewise, if we could relax this assumption, say by providing higher atomicity to process 0, we could reuse most of the proof.

8 Conclusion and Future Work

In this paper, we presented a component-based proof of Dijkstra's self-stabilizing token program that has been verified in PVS. To prove correctness of this self-stabilizing program, we needed to show two properties: (1) in the absence of faults, the program circulates a token along the ring, and (2) in the presence of faults, the program eventually recovers to a state from where the token circulation is restored. Following our philosophy of program decomposition, we decomposed the fault-tolerant program into the corresponding fault-intolerant program and the corrector. Then, we proved that property (1) is satisfied by focusing on the fault-intolerant program, and considering its execution starting from the invariant states. Subsequently, we proved property (2) by focusing on the corrector, and considering its execution starting from all states. Finally, we showed that the fault-intolerant program and the corrector do not interfere with each other.

Our case study illustrates that the advantages of program decomposition in non-mechanical proofs also apply to mechanical verification. It shows that by focusing on the component responsible for satisfying the property at hand, the proof of the required property is simplified. Also, it shows that the component-based approach readily supports design exploration as modifications to a program often permits the reuse of proofs. Moreover, it demonstrates that mechanical verification of fault-tolerant programs is less of a tour-de-force and more of a straightforward activity.

Regarding future work, we plan to investigate whether other techniques such as phased reasoning [14] based on convergence stairs [8] and hierarchical design of components offer the same advantage in mechanical verification as they do in non-mechanical verification. We also plan to investigate the use of program decomposition in mechanical verification of multitolerant programs [2], i.e., programs that tolerate multiple types of faults with possibly a different type of tolerance to each fault. Multitolerant programs can be decomposed into a fault-intolerant program and components responsible for tolerating each type of fault. Thus, the proof of tolerance property to a given type of fault can be simplified by focusing only on the components responsible for providing tolerance to that type of fault.

References

[1] A. Arora. *A foundation of fault-tolerant computing.* PhD thesis, The University of Texas at Austin, 1992.

[2] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[3] A. Arora and S. S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. *International Conference on Distributed Computing Systems*, pages 436–443, May 1998. An extended version of this paper is submitted to IEEE Transactions on Computers.

[4] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, 1988.

[5] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.

[6] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[7] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[8] M. G. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[9] Y. Lakhnech and M. Siegel. Deductive verification of stabilizing systems. *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 201–216, 1997.

[10] S. Merz. Mechanical verification of self-stabilizing token ring. Personal communication, 1998.

[11] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[12] I. S. W. B. Prasetya. Mechanically verified self-stabilizing hierarchical algorithms. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), volume 1217 of Lecture Notes in Computer Science*, pages 399–415, 1997.

[13] S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paul de Roever, editors, *IFIP International Conference on Programming Concepts and Methods (PROCOMET '98)*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.

[14] M. Siegel and F. Stomp. Extending the limits of sequentially phased reasoning. In P. S. Thiagarajan, editor, *Foundations of software technology and theoretical computer science, Lecture Notes in computer science 880*, 1994.

[15] G. Varghese. *Self-stabilization by local checking and correction.* PhD thesis, MIT/LCS/TR-583, 1993.

Symbols

Variable	Used as
p, q	program
$s, s0, s1, s2$	state
seq	trace
S, T	assertion
R	property
m, n	natural number
j, k	process, domain $0..N$
$v, v1, v2$	x value for a process, domain $0..(M-1)$

Expression	Meaning
$x(s)(j)$	The value of $x.j$ in state s
$seq(n)$	n^{th} state in the sequence seq

To be presented at the The 3rd Workshop on Human Error, Safety, and System Development (HESSD'99), Liege, Belgium, 7–8 June 1999.

Using Model Checking to Help Discover Mode Confusions and Other Automation Surprises*

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Abstract

Automation surprises occur when an automated system behaves differently than its operator expects. If the actual system behavior and the operator's "mental model" are both described as finite state transition systems, then mechanized techniques known as "model checking" can be used automatically to discover any scenarios that cause the behaviors of the two descriptions to diverge from one another. These scenarios identify potential surprises and pinpoint areas where design changes, or revisions to training materials or procedures, should be considered. The mental models can be suggested by human factors experts, or can be derived from training materials, or can express simple requirements for "consistent" behavior. The approach is demonstrated by applying the Murϕ state exploration system to a "kill-the-capture" surprise in the MD-88 autopilot.

This approach does not supplant the contributions of those working in human factors and aviation psychology, but rather provides them with a tool to examine properties of their models using mechanized calculation. These calculations can be used to explore the consequences of alternative designs and cues, and of systematic operator error, and to assess the cognitive complexity of designs.

The description of model checking is tutorial and is hoped to be accessible to those from the human factors community to whom this technology may be new.

1 Introduction

Automated systems sometimes behave in ways that surprise their operators [14]. These "automation surprises" are particularly well-documented in the cockpits of advanced commercial aircraft [10, 13] and several fatal crashes and other incidents are attributed to problems in the "flightcrew-automation interface" [6, Appendix D].

*This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931.

Norman [9] proposed that operators and users of automated systems form "mental models" of the way their system behaves and use these to guide their interaction with the system; an automation surprise can occur when the actual behavior of a system departs from its operator's mental model. Complex systems are often structured into "modes" (for example, an aircraft flight management system might have different modes for cruise, initial descent, landing, and so on), and their behavior can change significantly across different modes. "Mode confusion" arises when the system is in a different mode than that assumed by its operator; this is a rich source of automation surprises, since the operator may interact with the system according to a mental model that is inappropriate for its actual mode.

If we accept that automation surprises may be due to a mismatch between the actual behavior of a system and the operator's mental model of that behavior, then one way to look for potential surprises is to construct explicit descriptions of the actual system behavior, and of a postulated mental model, and to compare them. The discrete behavior of complex control systems can be specified by a "state machine," which is a formal, mathematical description that is amenable to various kinds of automated analysis. It is becoming accepted that such formal specifications can be useful in requirements analysis and other verification and validation activities for critical systems [3]. If a state machine specification is available for the actual system, and if we can construct one for a plausible mental model, then we could, in principle, "run" the two machines in parallel to see if their behaviors ever diverge from one another. What is potentially valuable about this approach is that if the two state machines have finite state spaces, then a body of techniques from the branch of formal methods known as "model checking" can be used to compare *all possible* behaviors of the two machines. If a discrepancy is discovered in the behaviors of the two system descriptions, a trace can be presented that gives the sequence of inputs and interactions that manifests the divergence. This provides the designer or analyst with information that can be used to bring the design of the actual system into closer alignment with the mental model (either by changing its behavior, or by improving the cues it provides to its operator), or that can be used to guide the formation of more appropriate mental models through improved operator training.

There are some obvious difficulties with this approach: the state machine descriptions of real systems often are not finite-state, or have finite state spaces that are too large to analyze exhaustively (this may be so, for example, if the state includes numeric quantities); also, there is no direct way to access an operator's mental model for the purpose of encoding it as a state machine. I am of the opinion that both these difficulties can be overcome by *abstraction* and *generalization*. Because we are performing refutation rather than verification (i.e., we are looking for potential bugs—automation surprises in this case—not trying to prove their absence) we do not need to model *all* the details of the actual system (e.g., to examine mode confusion, we need only model the mode transitions of the system, not the details of its behavior within those modes), so an abstracted description of the actual system that suppresses such details will be adequate. Also, we are not seeking psychological insight and do not need to examine the mental model of any particular operator—we

will be content to check whether the actual behavior violates plausible models and natural expectations (e.g., as suggested by training materials), and those concerned with developing, analyzing, documenting, and using the system should be able to guide construction of suitably generalized mental models. There is ample evidence from other applications of model checking that we learn more by examining *all* the behaviors of such approximated and generalized system descriptions than we do by examining just *some* of the behaviors of the real thing (as with simulation or direct testing).

2 An Example Scenario

I describe the proposed method using an example reported by Palmer [10, Case 2]. This example has also been analyzed by Leveson and Palmer [7]; I compare their approach with mine in Section 4.

The example is one of five altitude deviation scenarios observed during a NASA study in which twenty-two airline crews flew realistic two hour missions in DC-9 and MD-88 aircraft simulators. To follow the scenario, it is sufficient to understand that the autopilot can be instructed to cause the aircraft to climb or to hold a certain altitude through the setting of its "pitch mode." In VERT SPD (Vertical Speed) mode the aircraft climbs at the rate set by the corresponding dial (e.g., 2,000 feet per minute); in IAS (Indicated Air Speed) mode, it climbs at whatever rate is consistent with holding the air speed set by another dial (e.g., 256 knots); in ALT HLD (Altitude Hold) mode, it holds the current altitude. In addition, certain "capture modes" may be *armed*. If ALT (Altitude) capture is armed, the aircraft will only climb as far as the altitude set by the corresponding dial, at which point the pitch mode will change to ALT HLD; if the capture mode is not armed, however, and the pitch mode is VERT SPD or IAS, then the aircraft will continue climbing indefinitely. The behavior of this system is complicated by the existence of an ALT CAP (Altitude Capture) pitch mode, which is intended to provide smooth leveling off at the desired altitude. The ALT CAP pitch mode is entered automatically when the aircraft gets close to the desired altitude and the ALT capture mode is armed (do not confuse the ALT CAP *pitch* mode with the ALT *capture* mode). The ALT CAP pitch mode disarms the ALT capture mode and causes the plane to level off at the desired altitude, at which point it enters ALT HLD pitch mode.

The following scenario description is slightly reworded from Palmer's original in order to fit my terminology.

> The crew had just made a missed approach and had climbed to and leveled at 2,100 feet. They received the clearance to "...climb now and maintain 5,000 feet..." The Captain set the MCP (Master Control Panel) altitude window to 5,000 feet (causing ALT capture mode to become armed), set the autopilot pitch mode to VERT SPD with a value of approximately 2,000 ft. per minute and the autothrottle to SPD mode with a value of 256 knots. Climbing through 3,500 feet the Captain called for flaps up and at 4,000 feet he called for slats retract.

> Passing through 4000 feet, the Captain pushed the IAS button on the MCP. The pitch mode became IAS and the autothrottles went to CLAMP mode. The ALT capture mode was still armed. Three seconds later the autopilot automatically switched pitch mode to ALT CAP. The FMA (Flight Mode Annunciator) ARM window went from ALT to blank and the PITCH window showed ALT CAP. A tenth of a second later, the Captain adjusted the vertical speed wheel to a value of about 4,000 feet a minute. This caused the pitch autopilot to switch modes from ALT CAP to VERT SPD. As the altitude passed through 5,000 feet at a vertical velocity of about 4,000 feet per minute, the Captain remarked, "Five thousand. Oops, it didn't arm." He pushed the MCP ALT HLD button and switched off the autothrottle. The aircraft then leveled off at about 5,500 feet as the "*altitude—altitude*" voice warning sounded repeatedly.

An aircraft climbing through its assigned altitude (and potentially into the airspace assigned to another aircraft) is colloquially called a "bust," so Palmer refers to the scenario above as the "kill-the-capture bust." However, the basic problem is present whether or not it leads to a bust, so I prefer to speak of it as the "kill-the-capture surprise." The source of the surprise is the interaction of the pitch and capture modes and, in particular, with the way the ALT CAP pitch mode disarms the ALT capture mode. When the ALT capture mode is armed, changing the pitch mode between IAS and VERT SPD, or changing the values set by their corresponding dials, simply changes *how* the plane climbs to the desired altitude. When the aircraft gets close to the desired altitude, however, it autonomously enters ALT CAP pitch mode and disarms ALT capture mode. If the pitch mode is then changed to IAS or VERT SPD, the aircraft will climb without limit in the newly selected mode, since the ALT capture mode is now disarmed. The only indication to the pilot that the autopilot is in this vulnerable combination of modes is that the ARM window of the FMA changes from ALT to blank.

3 Analyzing the Example

To see how model checking techniques could reveal the existence of the kill-the-capture surprise, we first need to construct a mental model that a pilot might plausibly employ. Different pilots might have different mental models, and we cannot know what they are, but a plausible basic tenet might be that the pitch mode controls *how* the aircraft climbs, and the capture mode controls whether there is a *limit* to the climb. Another plausible basic tenet is that once capture mode is armed, it becomes disarmed only when the aircraft reaches the desired altitude (unless the pilot manually disarms it). Since this mental model makes no mention of the ALT CAP pitch mode, it obviously differs from the real system. This does not necessarily mean that the system harbors a surprise, however, because a mental model *should* suppress details considered unnecessary to understanding how to operate the system. The pilot might well be aware of the ALT CAP pitch mode and of its role in leveling

the plane off—and may even be aware that the ALT CAP pitch mode and the ALT capture mode interact in some way—but could believe this is merely the implementation of the ideal capture mode assumed in the mental model. To discover whether a surprise really does reside here, we need to "run" the state machines representing the actual system and the mental model on all possible sequences of inputs and compare their behavior.

I now present an automated analysis of this example using the Murϕ (pronounced "Murphy") state exploration system developed by David Dill's group at Stanford University [5]. Strictly speaking, Murϕ is not a model checker (that term is properly reserved for tools that test whether a transition system is a Kripke model for some temporal logic formula [2]), but the term "model checking" is loosely applied to any tool that uses (explicit or symbolic) state exploration techniques. Systems are described in Murϕ by specifying their *state variables*, and a series of *rules* that indicate the actions that can be performed by the system and the circumstances under which they can be performed. Properties that should hold in some or all states can be given as part of a Murϕ specification (as *assertions* and *invariants*, respectively), and the Murϕ system undertakes a search of all reachable states to ensure that the given properties do indeed hold. If they do not, Murϕ prints an error trace that describes the circumstances leading to the violation. Those who have some familiarity with computer programming should find it fairly easy to interpret Murϕ specifications and can think of Murϕ as performing exhaustive simulation of the specified system, so that *all possible* behaviors are examined; this is feasible because the number of states (i.e., combinations of values of the system variables) is finite (although it may be very large). In hardware and protocol applications, it is routine to apply Murϕ to specifications that are thousands of lines long and that have tens of millions of reachable states.

At the level of abstraction appropriate for our investigation, the actual behavior of the example system can be described in terms of two state variables, `pitch_mode` and `capture_armed`, which are specified in Murϕ as follows.

```
Type
  pitch_modes: enum {vert_speed, ias, alt_cap, alt_hold};
Var
  pitch_mode: pitch_modes;
  capture_armed: boolean;
```

These declarations specify that `pitch_mode` can take one of the four values from the enumerated type `pitch_modes`, and that `capture_armed` is a `boolean`. The `pitch_mode` state variable represents the autopilot's pitch mode in a direct way,[1] while the `capture_armed` variable encodes whether the ALT capture mode is armed. The initial state of the system is specified in the Murϕ `Startstate` declaration as follows.

[1] I use slightly different names to distinguish the pitch modes of the Murϕ model from those used in the narrative description, but the intended correspondence should be obvious.

```
Startstate
Begin
  clear pitch_mode;
  capture_armed := false;
End;
```

The `clear` construct chooses some arbitrary initial value.

Now we can specify the actions of the system by means of Murϕ rules as follows.

```
Rule "IAS"
Begin
  pitch_mode := ias;
End;
```

This rule corresponds to the pilot engaging the IAS pitch mode (whether by pushing its button, or entering a value in its dial is unimportant at this level of abstraction). It has no guards, meaning that it can "fire" at any time, and has the effect of setting the `pitch_mode` state variable to the value `ias`. The string `IAS` is simply the name used to identify the rule.

The HLD and VSPD rules are similar and correspond to the pilot engaging the ALT HLD and VERT SPD pitch modes, respectively.

```
Rule "HLD"
Begin
  pitch_mode := alt_hold;
End;

Rule "VSPD"
Begin
  pitch_mode := vert_speed;
End;
```

Notice that I do not model the parameters (e.g., speed, climb rate) used by the various pitch modes, nor the dials that are used to set these parameters. We are concerned only with the basic mode transitions, so it is appropriate to omit these details. I should also note that I have no idea whether the specification being developed here accurately represents the real DC-9 or MD-88 autopilots—my purpose is only to explain the approach, not to present an industrial application.

The following rule corresponds to the pilot pushing the ALT capture mode button. I have chosen to specify it as a toggle: initially the mode is not armed, pushing the button arms it, and pushing it again disarms it once more.

```
Rule "ALT CAPTURE"
Begin
  capture_armed := !capture_armed;
End;
```

The next rule corresponds to the aircraft approaching the selected altitude. I call the rule `near` and use lower case to distinguish it from the upper case names used for the rules associated with pilot actions that were presented above.

This rule only has an effect when `capture_armed` is `true`, in which case it sets `pitch_mode` to `alt_cap` and `capture_armed` to `false`. (Those familiar with Murϕ might wonder why I did not use `capture_armed` as a guard on the rule; the reason is that I will later need to modify the rule to incorporate the mental model and the present arrangement is more convenient for this purpose.)

```
Rule "near"
Begin
  If capture_armed Then
    pitch_mode := alt_cap;
    capture_armed := false;
  Endif;
End;
```

The next rule corresponds to the aircraft reaching the selected altitude when the pitch mode is ALT CAP, thereby causing a transition to ALT HLD. I originally specified this as follows,

```
Rule "arrived"
Begin
  If pitch_mode = alt_cap Then
    pitch_mode := alt_hold;
  Endif;
End;
```

However, we also need to account for the possibility that the pilot arms ALT capture mode when the aircraft is already at the selected altitude. This circumstance is dealt with by the second `If-Then` clause of the following revised rule, which disarms the ALT capture mode and bypasses ALT CAP to enter the ALT HLD pitch mode directly. In this and in later specification fragments, faint type is used for parts presented previously, and dark type for the new or changed material.

```
Rule "arrived"
Begin
  If pitch_mode = alt_cap Then
    pitch_mode := alt_hold;
  Endif;
  If capture_armed Then
    capture_armed := false;
    pitch_mode := alt_hold;
  Endif;
End;
```

Some readers may consider the specification of the last two rules to be excessively loose: for example, there is nothing in the specification that excludes physically impossible sequences of events, such as `arrived` followed by `near`, or several `near`s in succession. This looseness is typical in model checking: by omitting to specify constraints that are enforced by the physical world, or by other components of the system, we allow the specified system to have *more* behaviors than is actually possible. If this less constrained description does not exhibit the flaws we are concerned about, then certainly a more tightly specified system (having strictly fewer behaviors) will not exhibit them.[2] Only if we get "false drops" (i.e., apparent errors that would be excluded if the model was more detailed) will we need to refine the model.

We have now specified the behavior of the actual system and can turn to the specification of an idealization that constitutes a plausible mental model. A suitable model could be one where reaching the desired altitude causes ALT capture mode to be turned off and the pitch mode to change to ALT HLD; the `near` event is not significant to this mental model.

To specify this, I begin by adding a `boolean` state variable called `ideal_capture` that will record the state of the altitude capture mode in the mental model. This variable is initialized to `false` in the modified `Startstate` shown below.

```
Var
  pitch_mode: pitch_modes;
  capture_armed: boolean;
  ideal_capture: boolean;

Startstate
Begin
  clear pitch_mode;
  capture_armed := false;
  ideal_capture := false;
End;
```

The ideal capture mode is toggled by the ALT capture mode button in the same way as the arming of the real mode, so I add this to the specification of the `ALT CAPTURE` rule.

```
Rule "ALT CAPTURE"
Begin
  capture_armed := !capture_armed;
  ideal_capture := !ideal_capture;
End;
```

The ideal capture mode is unaffected by the `near` event, so that rule is left unchanged. If an `arrived` event occurs when the ideal capture mode is armed, then the mode is disarmed. This is specified by adding a third `If-Then` clause to the corresponding rule as follows.

[2]This is true for what are technically called *safety* properties; it is not true of *liveness* properties. All the properties considered here are safety properties.

```
Rule "arrived"
Begin
  If pitch_mode = alt_cap Then
    pitch_mode := alt_hold;
  Endif;
  If capture_armed Then
    pitch_mode := alt_hold;
    capture_armed := false;
  Endif;
  If ideal_capture Then
    ideal_capture := false;
  Endif;
End;
```

We now need to relate the ideal capture mode of the mental model to the modes of the actual system. The actual system is set to capture the desired altitude if *either* the pitch mode is ALT CAP *or* the capture mode is ALT. In terms of the Murϕ model this condition is given by the expression

```
(capture_armed | pitch_mode = alt_cap).
```

The modes of the actual system and of the mental model are consistent with each other if this expression is `true` exactly when `ideal_capture` is also `true`. We can state this in a Murϕ *invariant* as follows.

```
Invariant ideal_capture = (capture_armed | pitch_mode = alt_cap);
```

At this point, we have constructed specifications for the mode transitions of the actual system and of the mental model and stated, as an invariant, the condition for these to be consistent with each other. We can now proceed to examine whether any sequence of events can violate the invariant by causing Murϕ to perform exhaustive exploration of all the reachable states of the specification. Murϕ does this by systematically firing the rules of the specification in different orders until either an error is found or all possible cases have been examined. In this example, we receive the error trace shown in Figure 1.

This is exactly the scenario that manifested the automation surprise described in the previous section: the pilot engages the ALT capture mode, the aircraft approaches the desired altitude and automatically disarms the capture mode and engages the ALT CAP pitch mode, and then the pilot engages VERT SPD pitch mode. At this point the ideal capture mode is still armed, but that of the actual system is not. Murϕ found this scenario in 0.24 seconds (on a 400 MHz Pentium II with 256 MB of memory running Linux).

```
The following is the error trace for the error:

        Invariant "Invariant 0" failed.

Startstate Startstate 0 fired.
pitch_mode:vert_speed
capture_armed:false
ideal_capture:false
----------
Rule ALT CAPTURE fired.
capture_armed:true
ideal_capture:true
----------
Rule near fired.
pitch_mode:alt_cap
capture_armed:false
----------
Rule VSPD fired.
The last state of the trace (in full) is:
pitch_mode:vert_speed
capture_armed:false
ideal_capture:true
----------

End of the error trace.
```

Figure 1: First Error Trace

Leveson and Palmer also detected the potential for this surprise using their method [7] (I discuss the differences between their method and mine in the following section), and suggested that it could be eliminated by making two changes to the actual system. (My specification is organized differently to theirs, so the following translates the intent of their changes into the terms of my specification.)

- Cause the `arrived` event to engage ALT HLD pitch mode when the ALT capture mode is armed (as opposed to when the pitch mode is ALT CAP), and
- Cause disarming of ALT capture mode to occur when the pitch mode becomes ALT HLD rather than ALT CAP.

The intuition is that the ALT CAP pitch mode should be regarded as engaging a particular control law that determines *how* the aircraft flies the capture trajectory, but the ALT capture mode stays in effect until the desired altitude is achieved.

The first of the changes above is accomplished in our specification by deleting the first `If-Then` clause in the `arrived` rule, so that it becomes the following (I use a strikeout like ~~this~~ to indicate text that is removed).

```
Rule "arrived"
Begin
  ~~If pitch_mode = alt_cap Then~~
    ~~pitch_mode := alt_hold;~~
  ~~Endif;~~
  If capture_armed Then
    pitch_mode := alt_hold;
    capture_armed := false;
  Endif;
  If ideal_capture Then
    ideal_capture := false;
  Endif;
End;
```

The second change requires `capture_armed := false` to be removed from all rules that contain the assignment `pitch_mode := alt_cap` and added to all rules that contain the assignment `pitch_mode := alt_hold`. The `arrived` rule as modified above already satisfies this condition, but the HLD rule must be changed as follows.

```
Rule "HLD"
Begin
  pitch_mode := alt_hold;
  capture_armed := false;
End;
```

And the `near` rule must be changed to the following.

```
Rule "near"
Begin
  If capture_armed Then
    pitch_mode := alt_cap;
    ~~capture_armed := false;~~
  Endif;
End;
```

If we cause Murϕ to perform state exploration on this modified specification we obtain the error trace shown in Figure 2, which highlights a potential surprise introduced by the changes just made to the specification: if the pilot engages ALT HLD pitch mode while ALT capture mode is armed, the modified actual system will disarm the capture mode, while it remains armed in the mental model (and remained so in the actual system prior to the change). Inspection of Leveson and Palmer's specification indicates that this issue is present in their specification also, and is not just an artifact of my encoding. Several interpretations

```
The following is the error trace for the error:

        Invariant "Invariant 0" failed.

Startstate Startstate 0 fired.
pitch_mode:vert_speed
capture_armed:false
ideal_capture:false
----------
Rule ALT CAPTURE fired.
capture_armed:true
ideal_capture:true
----------
Rule HLD fired.
The last state of the trace (in full) is:
pitch_mode:alt_hold
capture_armed:false
ideal_capture:true
----------

End of the error trace.
```

Figure 2: Second Error Trace

seem plausible and reasonable for the intended behavior (and I have no idea what happens in this circumstance on a real aircraft), so we could modify either the description of the actual system, or that of the mental model, or both. I choose to suppose that ALT HLD pitch mode causes the aircraft to hold the current altitude, but that it should mask rather than disarm ALT capture mode—which will become active again if the pitch mode is changed to IAS or VERT SPD. This is consistent with the current mental model, and the prior system model, so the description of the actual system should be changed by undoing the change just made to the HLD rule (the other changes remain in place). This revision to the specification produces yet another error trace, shown in Figure 3.

This highlights yet another potential surprise in our specification: if the pilot presses the ALT button to arm the ALT capture mode and later, but before the desired altitude has been achieved, presses it again, the mental model indicates that the capture mode will be disarmed. This will be true of the actual system if the second button press occurs before the aircraft is near enough to the desired altitude to engage the ALT CAP pitch mode. But if the second button press occurs after ALT CAP mode has been engaged, then the actual system

```
The following is the error trace for the error:

        Invariant "Invariant 0" failed.

Startstate Startstate 0 fired.
pitch_mode:vert_speed
capture_armed:false
ideal_capture:false
----------
Rule ALT CAPTURE fired.
capture_armed:true
ideal_capture:true
----------
Rule near fired.
pitch_mode:alt_cap
----------
Rule ALT CAPTURE fired.
The last state of the trace (in full) is:
pitch_mode:alt_cap
capture_armed:false
ideal_capture:false
----------

End of the error trace.
```

Figure 3: Third Error Trace

does indeed disarm the ALT capture mode, but the aircraft will still be in the ALT CAP pitch mode, and hence still flying a capture trajectory.[3]

The best resolution to this issue is not obvious, so for simplicity I simply add a guard to the ALT CAPTURE rule that will cause ALT button presses to be ignored when the pitch mode is ALT CAP.

[3]This surprise is present, in a different form, in the original specification as well: if the ALT capture mode button is pressed after ALT CAP pitch mode has been engaged, then the original specification will arm the ALT capture mode (since it will have been disarmed when ALT CAP pitch mode was entered), but disarm the ideal capture mode.

Leveson and Palmer's specification uses a "push-pull," rather than a toggle arrangement for the ALT capture mode button, so this issue does not arise in their specification. However, I suspect that something like it must occur because their button seems to hold a state (i.e., "pushed in" or "pulled out") that is not synchronized with the internal system state.

```
Rule "ALT CAPTURE" pitch_mode != alt_cap ==>
Begin
  capture_armed := !capture_armed;
  ideal_capture := !ideal_capture;
End;
```

With this change, we finally bring the behaviors of the actual system and the mental model into alignment; Murϕ confirms this as shown in Figure 4.

```
Status:

        No error found.

State Space Explored:

        7 states, 41 rules fired in 0.23s.

Rules Information:

        Fired 7 times   - Rule "arrived"
        Fired 7 times   - Rule "near"
        Fired 7 times   - Rule "VSPD"
        Fired 7 times   - Rule "IAS"
        Fired 7 times   - Rule "HLD"
        Fired 6 times   - Rule "ALT CAPTURE"
```

Figure 4: Murϕ Reports Success

The output displays of the system have not been considered in the treatment presented so far. The quality of information presented to the operator is a critical factor in reducing automation surprises and mode confusion, and should certainly be examined in any comprehensive analysis. As a final illustration, I will indicate how this can be done using the model checking approach: the information displayed will be specified as part of the system description, the way it used by the operator will be part of mental model, and the interaction of these elements will be examined as part of the automated analysis.

An operator does not have access to all the data available to the actual system, and hence may not always know when a circumstance arises that calls for a mode change. Well-designed automation should keep the operator informed of these circumstances through its output displays. In addition, operators have limited memory and attention span and should not be expected to retain the internal state of their mental model infallibly. Good output displays should provide information that allows operators to "reload" their mental state. We can model an occasionally forgetful operator by adding a "whoops" rule to our specification as follows.

```
Rule "whoops"
Begin
  ideal_capture := !ideal_capture;
End;
```

This rule flips the value of `ideal_capture` and is invoked nondeterministically to model an operator who not merely forgets the state of his mental model, but "misremembers" the wrong one. Obviously, Murϕ detects numerous errors when this rule is added to the model without further adjustments.

Let us suppose, however, that the actual system turns on a light exactly when ALT capture mode is armed. The pilot's method of operation is changed so that, before performing any operation, she sets the state of the ideal capture mode of her mental model to be that indicated by the light. This is specified by adding the assignment `ideal_capture := capture_armed` to the beginning of the rules that represent pilot actions—namely, IAS, VSPD, HLD, and ALT CAPTURE.[4] Murϕ will again find that the `Invariant` fails in numerous circumstances (e.g., following the `whoops` rule). However, the only time it is really important for the actual system and the mental model to be in agreement is *following* any action by the pilot (so that the pilot can accurately predict the consequences of her actions). This can be accomplished by replacing the `Invariant` (which is evaluated after *every* rule) by `Assert` statements in the bodies of the four "pilot action" rules, as shown in Figure 5.

Murϕ reports no errors in this modified specification. (It is not hard to see by inspection that this must be so.) Additional experimentation will reveal that the guard on the ALT CAPTURE rule is still required, and that the only time `ideal_capture` does depart from the actual system state is in the `near` event when this follows a `whoops`. We regard this as unimportant, because it does not lead to a surprise in any action performed by the pilot. Combining this analysis with earlier ones, we conclude that the current design does not harbor surprises for a forgetful operator who follows the display light, nor for a nonforgetful one (independently of the light).

A notable property of all the analyses performed here is their simplicity and efficiency. Once the initial investment has been made to formalize the actual system behavior (and this might already have been done for other requirements analysis purposes), making adjustments to the system or mental model, performing state exploration, and examining the results is the work of minutes (none of the analyses described here took more than 0.25 seconds to run). Of course, the specifications used here have almost trivially small state spaces (from 7 to 14 states depending on the specification) and require very few rules to be fired (from 14 to 96). However, the evidence from other fields of application is that state exploration and model checking techniques scale quite well: it is routine to examine tens of millions of states with explicit enumeration, and often vastly more using symbolic methods.

[4]Because the light displays exactly the value of the state variable `capture_armed`, we do not need to introduce a new state variable or function to represent it.

```
Rule "IAS"
Begin
  ideal_capture := capture_armed;
  pitch_mode := IAS;
  Assert ideal_capture = (capture_armed | pitch_mode = alt_cap);
End;

Rule "VSPD"
Begin
  ideal_capture := capture_armed;
  pitch_mode := vert_speed;
  Assert ideal_capture = (capture_armed | pitch_mode = alt_cap);
End;

Rule "HLD"
Begin
  ideal_capture := capture_armed;
  pitch_mode := alt_hold;
  Assert ideal_capture = (capture_armed | pitch_mode = alt_cap);
End;

Rule "ALT CAPTURE" pitch_mode != alt_cap ==>
Begin
  ideal_capture := capture_armed;
  capture_armed := !capture_armed;
  ideal_capture := !ideal_capture;
  Assert ideal_capture = (capture_armed | pitch_mode = alt_cap);
End;
```

Figure 5: The "Pilot Action" Rules Modified to Use the Display Light

4 Discussion

There is much excellent work in the fields of system design, aviation psychology, ergonomics and human factors that seeks to understand and reduce the sources of operator error in automated systems. The work described here is intended to complement these existing studies by providing a practical, mechanized means to examine system designs for features that may be error prone. Human factors and other studies provide an idea of what to look for, and the work described here provides a method to look for it. The method uses existing tools for model checking and state exploration that have, in other kinds of applications, scaled successfully to quite large systems.

Model checking is a member of the class of techniques known as "formal methods," and there is also prior work, principally by Leveson and her colleagues, in applying formal

methods to the problems of automation surprises [8]. Leveson's work uses an evolving list of design features (currently there are about 15 items on the list) that are prone to cause operator mode awareness errors. These features provide criteria that can be applied to a formal system description in order to root out design elements that would repay additional consideration. Leveson and Palmer [7] apply this approach to the kill-the-capture surprise considered here. One of the error-prone design features identified by Leveson is use of "indirect" mode transitions which occur without explicit operator input. She and Palmer construct a formal specification of the relevant parts of the MD-88 autopilot and examine it (by hand) to detect such transitions. This approach successfully leads them to discover the indirect pitch mode transition to ALT CAP, and the confusing interaction between the pitch and capture modes. Leveson places great stress on the importance of precise and *readable* formal requirements specifications and has developed the SpecTRM methodology and its supporting requirements specification method SpecTRM-RL for this purpose. I am in full agreement on the importance of formal specifications that are readable as well as precise, and regard the work presented here as complementary to Leveson's in that it provides a way to automate analyses that she does by hand. Model checking techniques can be applied to any finite-state system description; I used Murϕ and its rather rebarbative notation simply because I am familiar with it, but it would certainly be feasible to develop comparable automation for SpecTRM-RL or other notations.

Automation is not a replacement for careful manual review of perspicuous, carefully structured formal specifications, but it is a valuable adjunct whose value becomes greater as the specifications get larger and their analysis correspondingly more difficult. The example considered here is almost trivially small, yet its automated analysis raised an issue that was not reported in Leveson and Palmer's manual examination—namely, that the repaired specification causes selection of the ALT HLD pitch mode to disarm the ALT capture mode. To be fair, Leveson and Palmer explicitly note that their repair to the kill-the-capture surprise "may violate other goals or desired behaviors of the autoflight system—the designers would have to determine this when deciding what solution to use. In addition, a more sophisticated solution may be required, e.g., a hysteresis factor may need to be added to the mode transition logic to avoid too rapid 'ping-ponging' transitions between pitch modes." Nonetheless, the fact remains that the approach used here found the original kill-the-capture surprise, found this issue with the repaired specification, and found another issue (namely that pressing the ALT capture mode button after the pitch mode has changed to ALT CAP does not disarm the altitude capture)—all with essentially no effort. It also allowed rapid and inexpensive exploration of an occasionally forgetful operator and of the efficacy of displays in mitigating this problem. The ability to use formal analysis in this manner for active design exploration is an underappreciated attribute of formal methods—and one that depends critically on efficiently mechanized methods of analysis.

Many authors have observed that model checking and other forms of automated formal analysis can usefully be applied to requirements specifications. Indeed, Leveson and Palmer propose that "the pilot's mental model includes a cause and effect relationship be-

tween arming the altitude capture and eventually... acquiring that altitude and holding it" and this phraseology almost immediately invites formulation in temporal logic (such logics provide an *eventually* modality), which is the classical application of model checking. A little thought and experimentation, however, reveals that it is generally difficult or impossible to formulate a mental model, or the expectations it engenders, within the limited expressivity of a temporal logic. In the example just quoted, it would be necessary to add the caveat "provided the pilot does not explicitly disarm altitude capture" and this is not easily stated in temporal logic. Furthermore, the suggested formulation relates a mode control issue ("arming the altitude capture") to an external event ("acquiring that altitude"). In order to examine this relationship, our formal model would need to include some treatment (e.g., qualitative physics) for the notion of an aircraft "climbing" and its relation to "altitude" that would add greatly to its complexity.

The novelty and utility in the approach used here is that it moves specification of the desired behavior from the property/assertion language of the model checker into its system specification language. That is to say, the desired property is conceived as a mental model that is specified as a state machine running in parallel with the state machine that specifies the actual system. This seems consistent with representations already employed in the human factors community [4], and provides the expressiveness needed to accommodate possibilities such as the pilot explicitly disarming altitude capture, while allowing the correctness criterion to be stated in terms of (idealized) modes rather than external physical realities (such as reaching a desired height). The property/assertion language of the model checker or state exploration system is used simply to state (as an invariant) the desired correspondence between actual and idealized modes.

In more technical terms, we are really checking a simulation relation between two system descriptions (the mental model and the actual system). This is a basic capability of "model checkers" for process algebras, such as the FDR tool for CSP [11], but must be achieved somewhat indirectly in tools based on state transition relations such as Murϕ. The approach used here works in simple cases; in more complicated cases, it may be necessary to use superposition and an explicit abstraction (or, dually, refinement) relation (see [12] for a tutorial explanation).

As noted earlier, in addition to global `invariants`, Murϕ also allows `assert` statements in the bodies of its rules; these provide a way of checking additional properties, such as those that should hold on mode *transitions* (as opposed to when the system is *in* a mode). For example, we can add an `assert` statement to the rule `arrived` to check that the pitch mode is indeed ALT HLD whenever the ideal capture mode is disarmed as a result of the `arrived` event.

```
Rule "arrived"
Begin
  If capture_armed Then
    pitch_mode := alt_hold;
    capture_armed := false;
  Endif;
  If ideal_capture Then
    ideal_capture := false;
    Assert pitch_mode = alt_hold;
  Endif;
End;
```

This check is satisfied (provided there are no whoops events) in the final specification presented here, but detects issues (that were also found by the `Invariant`) in earlier specifications.

The expressiveness provided by this approach opens a number of interesting possibilities for modeling and analysis in addition to those already illustrated.

- We can examine the consequences of a faulty operator: simply endow the operator model with selected faulty behaviors and observe their consequences. The effectiveness of remedies such as lockins and lockouts, or improved displays, can be evaluated similarly.

- We can examine the load placed on an operator: if the simplest mental model that can adequately track the actual system requires too many states, or a moderately complex data structure such as a stack, we can evaluate the reduction achieved by additional or improved output displays, or by redesigning the system behavior.

- We can examine the accuracy of an operator instruction manual by formulating it as a transition system and comparing it to a similar formulation of its actual system—just we formulated and compared a mental model with its actual system in the example.

In the future, I hope that the approach described here will be developed and documented further, and extended in the directions just listed. I also look forward to evaluating it on a more realistic example. It will also be interesting to compare this approach with one in which formal methods are used to examine similar specifications for consistency and safety properties expressed directly as invariants [1].

Acknowledgments

I am grateful for help and encouragement received from several people. Comments from Judy Crow improved the presentation of this material. David Dill suggested the forgetful operator and a number of the other extensions listed above. I received very helpful feedback from talks at NASA Ames and Langley Research Centers, and at Rockwell Collins.

References

[1] Ricky W. Butler, Steven P. Miller, James N. Potts, and Victor A. Carreño. A formal methods approach to the analysis of mode confusion. In *17th AIAA/IEEE Digital Avionics Systems Conference*, Bellevue, WA, October 1998.

[2] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[3] Judith Crow and Ben L. Di Vito. Formalizing Space Shuttle software requirements: Four case studies. *ACM Transactions on Software Engineering and Methodology*, 7(3):296–332, July 1998.

[4] Asaf Degani, Michael Shafto, and Alex Kirlik. Modes in automated cockpits: Problems, data analysis, and a modeling framework. In *Proceedings of the 36th Israel Annual Conference on Aerospace Sciences*, Haifa, Israel, 1996. Available at `http://olias.arc.nasa.gov/publications/degani/modeusage/modes2.html`.

[5] David L. Dill. The Murϕ verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, pages 390–393, New Brunswick, NJ, July/August 1996. Volume 1102 of *Lecture Notes in Computer Science*, Springer-Verlag.

[6] The interfaces between flightcrews and modern flight deck systems. Report of the FAA human factors team, Federal Aviation Administration, 1995. Available at `http://www.faa.gov/avr/afs/interfac.pdf`.

[7] Nancy G. Leveson and Everett Palmer. Designing automation to reduce operator errors. In *Proceedings of the IEEE Systems, Man, and Cybernetics Conference*, October 1997. Available at `http://www.cs.washington.edu/research/projects/safety/www/papers/smc.ps`.

[8] Nancy G. Leveson, L. Denise Pinnel, Sean David Sandys, Shuichi Koga, and Jon Damon Rees. Analyzing software specifications for mode confusion potential. In C. W. Johnson, editor, *Proceedings of a Workshop on Human Error and System Development*, pages 132–146, Glasgow, Scotland, March 1997. Glasgow Accident Analysis Group, technical report GAAG-TR-97-2. Paper available at `http://www.cs.washington.edu/research/projects/safety/www/papers/glascow.ps`.

[9] Donald A. Norman. *The Psychology of Everyday Things*. Basic Books, New York, NY, 1988. Also available in paperback under the title "The Design of Everyday Things.".

[10] Everett Palmer. "Oops, it didn't arm." A case study of two automation surprises. In Richard S. Jensen and Lori A. Rakovan, editors, *Proceedings of the Eighth International Symposium on Aviation Psychology*, pages 227–232, Columbus, OH, April 1995. The Aviation Psychology Laboratory, Department of Aerospace Engineering, Ohio State University. Paper available at `http://olias.arc.nasa.gov/~ev/OSU95_Oops/PalmerOops.html`.

[11] A. W. Roscoe. Model-checking CSP. In *A Classical Mind: Essays in Honour of C. A. R. Hoare*, Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, UK, 1994.

[12] John Rushby. Combining system properties: A cautionary example and formal examination. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, June 1995. Unpublished project report; available at `http://www.csl.sri.com/~rushby/combined.html`.

[13] N. B. Sarter and D. D. Woods. How in the world did we ever get into that mode? Mode error and awareness in supervisory control. *Human Factors*, 37(1):5–19, 1995.

[14] N. B. Sarter, D. D. Woods, and C. E. Billings. Automation surprises. In Gavriel Salvendy, editor, *Handbook of Human Factors and Ergonomics*. John Wiley and Sons, second edition, 1997.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Presented at the 18th AIAA/IEEE Digital Avionics Systems Conference (DASC), St Louis, MO, October 1999.

An Automated Method To Detect Potential Mode Confusions*

John Rushby, SRI International, Menlo Park, California

Judith Crow, SRI International, Menlo Park, California

Everett Palmer, NASA Ames Research Center, Moffett Field, California

Introduction

Mode confusions are a type of "automation surprise"—circumstances where an automated system behaves differently than its operator expects. It is generally accepted that operators develop "mental models" for the behavior of automated systems and use these to guide their interaction with the systems concerned, so that an automation surprise results when the actual system behavior diverges from its operator's mental model.

Complex systems are often structured into "modes" (for example, an autopilot might have different modes for altitude capture, altitude hold, and so on), and their behavior can change significantly across different modes. "Mode confusion" arises when the system is in a different mode than that assumed by its operator; this is a rich source of automation surprises, since the operator may interact with the system according to a mental model that is inappropriate for its actual mode. Mode confusions have been implicated in several recent crashes and other incidents, and are a growing source of concern in modern automated cockpits.

If we accept that mode confusions are due to a mismatch between the actual behavior of a system and the mental model of its operator, then one way to look for potential mode confusions is to compare the design of the actual system against a mental model. There are two challenges here: how to get hold of a mental model, and how to do the comparison.

*This work was supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under contract F49620-95-C0044 and by the National Science Foundation under contract CCR-9509931.

Through observation, questionnaires, and other techniques, psychologists have been able to elicit the mental models of individual operators (typically pilots). However, comparison between a design and the mental model of a specific individual will provide only very specific information; we are interested in whether a design is *prone* to mode confusions, and for this purpose it is more useful to compare the design against a generic mental model rather than that of an individual. Such a generic model can be extracted from training material (one of the purposes, often implicit, of a training manual is to induce adequate mental models) or specified as an explicit requirement (e.g., "this button should behave like a toggle"). Cognitive studies provide two important insights on the nature of these models: first, that they can be represented compactly by mathematical structures called "state machines"; second, that they tend to be fairly simple (which can be explained by application of two canonical simplifications [3]).

The fact that mental models can be represented as state machines suggests a solution to the second challenge mentioned above—for designs can also be represented as state machines (this idea underlies modern design techniques, such as Statecharts), and there are automated methods for comparing the behaviors of one (finite) state machine against those of another. These methods are members of a class of formal techniques, known as "model checking," that are quite mature and are used routinely in hardware design and protocol analysis to explore properties of systems having many millions of states.

An Example

We outline the proposed method using a "kill the capture" example reported by Palmer [6, Case 2].

The example is one of five altitude deviation scenarios observed during a NASA study in which twenty-two airline crews flew realistic two hour missions in DC-9 and MD-88 aircraft simulators. To follow the scenario, it is sufficient to understand that the autopilot can be instructed to cause the aircraft to climb or to hold a certain altitude through the setting of its "pitch mode." In VSP (Vertical Speed) mode the aircraft climbs at the rate set by the corresponding dial (e.g., 2,000 feet per minute); in IAS (Indicated Air Speed) mode, it climbs at whatever rate is consistent with holding the air speed set by another dial (e.g., 256 knots); in HLD (Altitude Hold) mode, it holds the current altitude. In addition, certain "capture modes" may be *armed*. If ALT (Altitude) capture is armed, the aircraft will only climb as far as the altitude set by the corresponding dial, at which point the pitch mode will change to HLD; if the capture mode is not armed, however, and the pitch mode is VSP or IAS, then the aircraft will continue climbing indefinitely. The behavior of this system is complicated by the existence of an ALT CAP (Altitude Capture) pitch mode, which is intended to provide smooth leveling off at the desired altitude. The ALT CAP pitch mode is entered automatically when the aircraft gets close to the desired altitude and the ALT capture mode is armed (do not confuse the ALT CAP *pitch* mode with the ALT *capture* mode). The ALT CAP pitch mode disarms the ALT capture mode and causes the plane to level off at the desired altitude, at which point it enters HLD pitch mode.

The following scenario description is slightly reworded from the original to fit the terminology used here.

> The crew had just made a missed approach and had climbed to and leveled at 2,100 feet. They received the clearance to "...climb now and maintain 5,000 feet..." The Captain set the MCP (Master Control Panel) altitude window to 5,000 feet (causing ALT capture mode to become armed), set the autopilot pitch mode to VSP with a value of approximately 2,000 ft. per minute and the autothrottle to SPD mode with a value of 256 knots. Climbing through 3,500 feet the Captain called for flaps up and at 4,000 feet he called for slats retract. Passing through 4000 feet, the Captain pushed the IAS button on the MCP. The pitch mode became IAS and the autothrottles went to CLAMP mode. The ALT capture mode was still armed. Three seconds later the autopilot automatically switched pitch mode to ALT CAP. The FMA (Flight Mode Annunciator) ARM window went from ALT to blank and the PITCH window showed ALT CAP. A tenth of a second later, the Captain adjusted the vertical speed wheel to a value of about 4,000 feet a minute. This caused the pitch autopilot to switch modes from ALT CAP to VSP. As the altitude passed through 5,000 feet at a vertical velocity of about 4,000 feet per minute, the Captain remarked, "Five thousand. Oops, it didn't arm." He pushed the MCP HLD button and switched off the autothrottle. The aircraft then leveled off at about 5,500 feet as the "*altitude—altitude*" voice warning sounded repeatedly.

To see how model checking techniques could reveal the existence of the surprise in this scenario, we first need to construct a mental model that a pilot might plausibly employ. A plausible generic model might embody the idea that the pitch mode controls *how* the aircraft climbs, and the capture mode controls whether there is a *limit* to the climb. Another plausible component of a generic model is that once capture mode is armed, it becomes disarmed only when the aircraft reaches the desired altitude (unless the pilot manually disarms it).

This mental model is described by the state machine in Figure 1, where HLD, ALT, IAS, and VSP represent the events where the corresponding buttons are pressed, and `arrive` represents the event of the aircraft reaching the target altitude. There are three states in this model, representing the (mutually exclusive) situations where the aircraft is in altitude hold, or has an altitude capture active, or neither of these. Pressing the IAS or VSP buttons has no effect (on

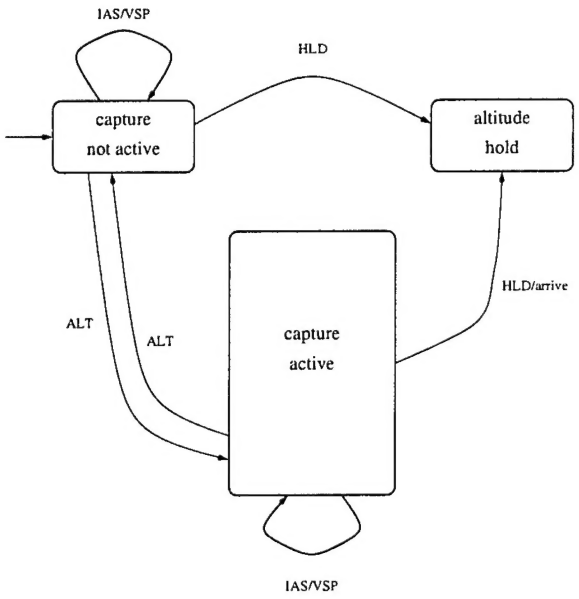


Figure 1: State Machine for Mental Model

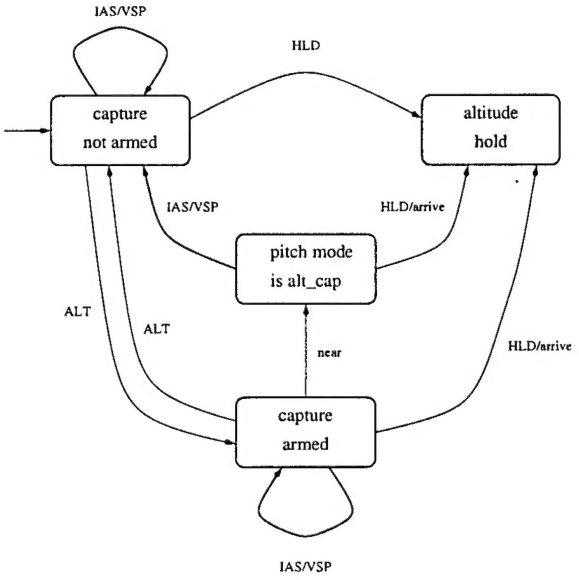


Figure 2: State Machine for Actual System

the aspects of behavior considered here) when in either of the latter two states; pressing ALT causes each of these states to transition to the other; and pressing the HLD button causes a transition into altitude hold; arriving at the desired altitude also cases a transition into altitude hold when a capture is active. (To keep things simple, we do not model other behaviors, such as pressing the HLD button when already in altitude hold.)

The actual system is rather more complicated than the mental model: it has an ALT CAP pitch mode that is entered autonomously when a capture is active and the aircraft gets near the desired altitude. This behavior (also somewhat simplified) is described by the state machine in Figure 2

Since the mental model makes no mention of the ALT CAP pitch mode, it obviously differs from the actual system. This does not necessarily mean that the system harbors a surprise, however, because a mental model *should* suppress details considered unnecessary to understanding how to operate the system. The pilot might well be aware of the ALT CAP pitch mode and of its role in leveling the plane off—and may even be aware that the ALT CAP pitch mode and the ALT capture mode interact in some way—but could believe this is merely the implementation of the ideal capture mode assumed in the mental model. To discover whether a surprise really does reside here, we need to "run" the state machines representing the actual system and the mental model on all possible sequences of inputs and compare their behavior.

In order to compare the behavior of the two state machines, we need to relate their states to each other. The actual system is flying a capture (in different ways) when in either of the two states in the lower center of Figure 2, so we can "abstract" these into a compound state that corresponds to the single "capture active" state of the mental model. This is shown in Figure 3.

Erasing the detail in the compound state, we arrive at the fully abstracted state machine for the actual system that is shown in Figure 4.

The states and events of this abstracted state machine correspond in the obvious way to those of the mental model, and any potential automation surprise will be manifested as a difference in their transition structures. Comparing Figures 1 and 4 we immediately see such a difference: the transition on VSP and IAS from the compound state of the abstracted actual system to the "capture not active" state. (Notice there are two transitions labelled IAS/VSP from the compound state; these represent nondeterministic choice at this level—we have to look inside the compound state to see which transition will actually occur.) This suggests that the sequence of events: VSP,

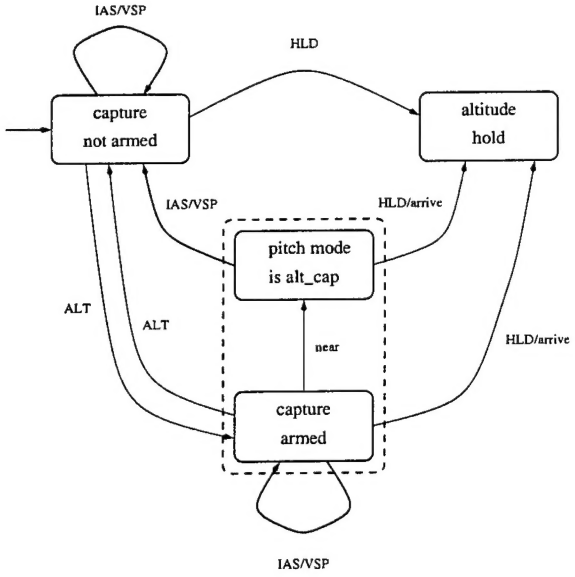


Figure 3: Partially Abstracted State Machine for Actual System

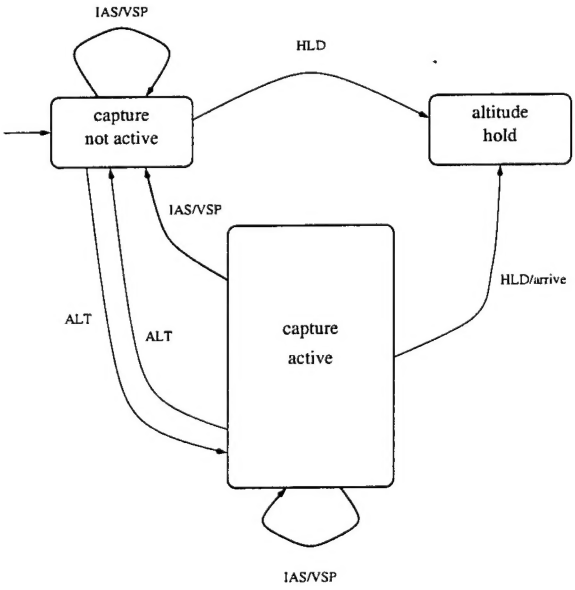


Figure 4: Abstracted State Machine for Actual System

`ALT, near, VSP` will lead to a surprise (capture will be active in the mental model but not in the actual system), and this is exactly the sequence that gave rise to the "kill the capture" scenario.

The diagrammatic approach used above is intended only to convey the intuition behind this method of analysis. To obtain a method suitable for practical application, we need the reliability and power of automated tools. Model checkers such as Murphi, SMV, and SPIN (which are all freely available) can provide this automation. To apply a model checker to these problems, we first specify the actual system and the mental model as state machines in the language of the tool concerned, then specify the expected abstraction relationship between the two descriptions, and cause the model checker to search for a violation of this relationship. Model checkers work by performing (explicit or symbolic) search over the entire state space of the state machines supplied to them; what makes them useful is that they can explore vast numbers of states (many millions) in a reasonable time. (Model checkers differ from simulators in that they consider all possible scenarios.) A companion paper [7] describes how the example considered here can be subjected to automated analysis using the model checker Murphi from David Dill's group at Stanford. The specification of the actual system and mental model requires just a few dozen lines of specification, and the automated analysis finds the "kill the capture" scenario in a fraction of a second. More interestingly, the analysis finds another surprise in a modified system specification that is intended to correct the source of the original surprise, and then another surprise in a further modification. The analysis was also extended to model the behavior of a faulty operator (who "misremembers" the current mode) and the utility of displays in correcting this.

Discussion

There is much excellent work in the fields of system design, aviation psychology, ergonomics and human factors that seeks to understand and reduce the sources of operator error in automated systems. The method described here is intended to complement these existing studies by providing a practical, mechanized means to examine system designs for features that may be error prone. Human factors and other studies provide an idea of what to look for, and the method described here provides a method to look for it. The method uses existing tools for model checking and state exploration that have, in other kinds of applications, scaled successfully to quite large systems.

Model checking is a member of the class of techniques known as "formal methods" and there has

been prior work in applying formal methods to the problems of automation surprises. For example, Leveson and colleagues [5] compare system designs (by hand) against a list of design features that are prone to cause operator mode awareness errors. One of the error-prone design features identified by Leveson is use of "indirect" mode transitions—those which occur without explicit operator input. This approach was applied to the example considered here by Leveson and Palmer [4] and successfully identified the indirect pitch mode transition to `ALT CAP` in the example considered here, and the confusing interaction between the pitch and capture modes. However, that manual analysis did not report the surprise in the modified system description that was detected by our automated analysis using Murphi.

In other work that applies automated formal methods to mode confusion, Butler and colleagues [1] examine an autopilot specification for satisfaction of consistency and safety properties expressed as invariants.

We see the method presented here as complementary to these other approaches: automation is an adjunct, not a replacement, for careful human review of design specifications, and checking for consistent behavior is surely desirable in any analysis of an interactive system. What our method contributes is the ability to include an explicit model of the operator in the analysis. This allows detection of potential mode confusions that are beyond the scope of these other methods, but depends on the construction of suitable mental models.

For the future, we plan to apply this method to larger examples, and to evaluate its effectiveness in more realistic applications. We are also interested in using the technique to explore the consequences of operator error (and the effectiveness of remedies such as lock-ins and lockouts, or improved displays), and to assess the cognitive load placed on an operator by a given design (e.g., if the simplest mental model that can adequately track the actual system requires a large number of states, or a data structure such as a stack, then we may conclude that the design is too complex). We are also interested in exploring potential interactions between multiple mental models (e.g., for different aspects of the behavior of a single system), the consequences of inappropriate mental models (e.g., the interaction between a model of normal operation and a system operating in an unusual mode), and in using this method to assess training materials and checklists.

References

[1] Ricky W. Butler, Steven P. Miller, James N. Potts, and Victor A. Carreño. A formal methods approach to the analysis of mode confusion. In *17th AIAA/IEEE Digital Avionics Systems Conference*, Bellevue, WA, October 1998.

[2] Denis Javaux and Véronique De Keyser, editors. *Proceedings of the 3rd Workshop on Human Error, Safety, and System Development (HESSD'99)*, University of Liege, Belgium, June 1999.

[3] Denis Javaux and Peter G. Polson. A method for predicting errors when interacting with finite state machines. In Javaux and Keyser [2].

[4] Nancy G. Leveson and Everett Palmer. Designing automation to reduce operator errors. In *Proceedings of the IEEE Systems, Man, and Cybernetics Conference*, October 1997. Available at `http://www.cs.washington.edu/research/projects/safety/www/papers/smc.ps`.

[5] Nancy G. Leveson, L. Denise Pinnel, Sean David Sandys, Shuichi Koga, and Jon Damon Rees. Analyzing software specifications for mode confusion potential. In C. W. Johnson, editor, *Proceedings of a Workshop on Human Error and System Development*, pages 132–146, Glasgow, Scotland, March 1997. Paper available at `http://www.cs.washington.edu/research/projects/safety/www/papers/glasco%w.ps`.

[6] Everett Palmer. "Oops, it didn't arm." A case study of two automation surprises. In Richard S. Jensen and Lori A. Rakovan, editors, *Proceedings of the Eighth International Symposium on Aviation Psychology*, pages 227–232, The Aviation Psychology Laboratory, Department of Aerospace Engineering, Ohio State University, Columbus, OH, April 1995. Paper available at `http://olias.arc.nasa.gov/~ev/OSU95_Oops/PalmerOops.html`.

[7] John Rushby. Using model checking to help discover mode confusions and other automation surprises. In Javaux and Keyser [2].

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.